

Guidewire PolicyCenter®

PolicyCenter Configuration Guide

RELEASE 8.0.3

Copyright © 2001-2014 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, ClaimCenter, BillingCenter, PolicyCenter, InsuranceSuite, 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: 8.0.3

Document Name: *PolicyCenter Configuration Guide*

Document Revision: 18-November-2014

Contents

About PolicyCenter Documentation	25
Conventions in This Document	26
Support	26

Part I PolicyCenter Configuration Basics

1 Overview of PolicyCenter Configuration	29
What You Can Configure	29
How You Configure PolicyCenter	30
Types of Application Environments	31
The Development Environment	31
The Production Environment	31
Deploying Configuration Files	31
Deploying Changes in a Development Environment	31
Deploying Changes to the Production Server	32
Regenerating the Data Dictionary and Security Dictionary	32
Generating the Data and Security Dictionaries in HTML Format	33
Generating the Data and Security Dictionaries in XML Format	33
Generating the Dictionaries as You Generate a .war or .ear File	33
Aspects of Regenerating the Security Dictionary	33
Managing Configuration Changes	34
2 Application Configuration Parameters	35
Working with Configuration Parameters	36
Accessing Configuration Parameters in Gosu	36
Configuration Parameter Attributes	37
Adding Custom MIME Types	37
Archive Parameters	38
ArchiveEnabled	38
ArchiveDaysRetrievedBeforeArchive	38
ArchiveDefaultRecheckDays	39
ArchivePolicyTermDays	39
ArchiveRecentJobCompletionDays	39
Assignment Parameters	39
AssignmentQueuesEnabled	39
Batch Process Parameters	39
BatchProcessHistoryPurgeDaysOld	39

Business Calendar Parameters	39
BusinessDayDemarcation	40
BusinessDayEnd	40
BusinessDayStart	40
BusinessWeekEnd	40
HolidayList (Obsolete)	40
IsFridayBusinessDay	40
IsMondayBusinessDay	40
IsSaturdayBusinessDay	41
IsSundayBusinessDay	41
IsThursdayBusinessDay	41
IsTuesdayBusinessDay	41
IsWednesdayBusinessDay	41
MaxAllowedDate	41
MinAllowedDate	41
Cache Parameters	41
ExchangeRatesCacheRefreshIntervalSecs	42
GlobalCacheActiveTimeMinutes	42
GlobalCacheDetailedStats	42
GlobalCacheReapingTimeMinutes	42
GlobalCacheSizeMegabytes	42
GlobalCacheSizePercent	43
GlobalCacheStaleTimeMinutes	43
GlobalCacheStatsRetentionPeriodDays	43
GlobalCacheStatsWindowMinutes	43
GroupCacheRefreshIntervalSecs	43
ScriptParametersRefreshIntervalSecs	43
TreeViewRefresh	44
ZoneCacheRefreshIntervalSecs	44
Clustering Parameters	44
ClusteringEnabled	44
ClusterMemberPurgeDaysOld	45
ClusterMemberRecordUpdateIntervalSecs	45
ClusterMulticastAddress	45
ClusterMulticastPort	45
ClusterMulticastTTL	45
ClusterProtocolStackOption1	46
ClusterProtocolStackOption2	46
ClusterProtocolStack	46
ClusterStatisticsMonitorIntervalMins	46
ConfigVerificationEnabled	46
JGroupsWatchdogHeartbeatIntervalSecs	47
JGroupsWatchdogMissedHeartbeatsBeforeReset	47
PDFMergeHandlerLicenseKey	47
Database Parameters	47
DisableHashJoinPolicySearch	47
DisableIndexFastFullScanForPolicySearch	47
DisableSortMergeJoinPolicySearch	47
DiscardQueryPlansDuringStatsUpdateBatch	48
IdentifyQueryBuilderViaComments	48
IdentifyORMLayerViaComments	48
MigrateToLargeIDsAndDatetime2	48

Desktop and Team Parameters	48
OtherWorkOrdersStatisticsWindowSize	49
RenewalsStatisticsWindowSize	49
SearchActivityThresholdDays	49
SubmissionsStatisticsWindowSize	49
TeamScreenTabVisibilityRowCountCutoff	50
Document Creation and Document Management Parameters	50
AllowDocumentAssistant	50
DisplayDocumentEditUploadButtons	50
DocumentAssistantJNLP	50
DocumentContentDispositionMode	51
DocumentTemplateDescriptorXSDLocation	51
MaximumFileUploadSize	51
UseDocumentAssistantToDisplayDocuments	51
Domain Graph Parameters	51
DomainGraphKnownLinksWithIssues	51
DomainGraphKnownUnreachableTables	52
Environment Parameters	52
AddressVerificationFailureAsError	52
AlwaysShowPhoneWidgetRegionCode	52
CurrentEncryptionPlugin	52
DeprecatedEventGeneration	53
EnableAddressVerification	53
EnableInternalDebugTools	53
KeyGeneratorRangeSize	53
MemoryUsageMonitorIntervalMins	53
PublicIDPrefix	54
ResourcesMutable	54
RetainDebugInfo	55
StrictDataTypes	55
TwoDigitYearThreshold	55
UnreachableCodeDetection	56
UnrestrictedUserName	56
UseOldStylePreUpdate	56
WarnOnImplicitCoercion	56
WebResourcesDir	56
Financial Parameters	56
Geocoding Feature Parameters	56
UseGeocodingInPrimaryApp	57
ProximitySearchOrdinalMaxDistance	57
ProximityRadiusSearchDefaultMaxResultCount	57
UseMetricDistancesByDefault	57
Globalization Parameters	57
DefaultApplicationLanguage	58
DefaultApplicationLocale	58
DefaultApplicationCurrency	58
DefaultRoundingMode	58
MulticurrencyDisplayStyle	59
DefaultCountryCode	59
DefaultPhoneCountryCode	59
DefaultNANPACountryCode	60
AlwaysShowPhoneWidgetRegionCode	60

Integration Parameters	60
BillingSystemURL	60
ClaimSystemURL	60
DefaultXmlExportEncryptionId	61
KeepCompletedMessagesForDays	61
LockPrimaryEntityDuringMessageHandling	61
PaymentSystemURL	61
PluginStartupTimeout	61
Job Expiration Parameters	62
JobExpirationCheckAudit	62
JobExpirationCheckCancellation	63
JobExpirationCreateDateThreshold	63
JobExpirationEffDateThreshold	63
JobExpireCheckIssuance	63
JobExpireCheckPolicyChange	63
JobExpireCheckReinstatement	63
JobExpireCheckRenewal	64
JobExpireCheckRewrite	64
JobExpireCheckSubmission	64
JobExpireCheckTestJob	64
Lookup Table Parameters	64
AvailabilityContextCacheSize	64
Miscellaneous Job-Related Parameters	64
AllowedDaysBeforeOrAfterPolicyStartDate	64
BoundPolicyThresholdDays	65
ClosedPolicyThresholdDays	65
MaximumPolicyCreationYearDelta	65
MaxRecentAccounts	65
MaxRecentPoliciesAndJobs	65
MaxSubmissionsToCreate	65
MinimumPolicyCreationYear	65
OpenPolicyThresholdDays	65
PatternCacheMaxDuration	66
PolicyChangeMaxQuotes	66
RenewalMaxQuotes	66
RenewalProcessLeadTime	66
SubmissionMaxQuotes	66
Miscellaneous Parameters	66
ActivityStatisticsWindowSize	66
ConsistencyCheckerThreads	66
DefaultDiffDateFormat	66
DisableDomainGraphSupport	67
IgnoreLeapDayForEffDatedCalc	67
InitialSampleDataSet	68
JGroupsClusterChannel	68
ListViewPageSizeDefault	68
ProfilerDataPurgeDaysOld	68
TransactionIdPurgeDaysOld	68

PDF Print Settings Parameters	68
DefaultContentDispositionMode	69
PrintFontFamilyName	69
PrintFontSize	69
PrintFOPUserConfigFile	69
PrintHeaderFontSize	69
PrintLineHeight	69
PrintListViewBlockSize	69
PrintListViewFontSize	70
PrintMarginBottom	70
PrintMarginLeft	70
PrintMarginRight	70
PrintMarginTop	70
PrintMaxPDFInputFileSize	70
PrintPageHeight	70
PrintPageWidth	70
Product Model	71
ExternalProductModelDirectory	71
Quote Purging Configuration Parameters	71
PruneAndPurgeJobsEnabled	71
PruneVersionsDefaultRecheckDays	72
PruneVersionsPolicyTermDays	72
PruneVersionsPolicyTermDaysCheckDisabled	72
PruneVersionsRecentJobCompletionDays	72
PurgeJobsDefaultRecheckDays	72
PurgeJobsPolicyTermDays	72
PurgeJobsPolicyTermDaysCheckDisabled	72
PurgeJobsRecentJobCompletionDays	73
PurgeOrphanedPolicyPeriodsEnabled	73
Rating Management Parameters	73
PurgeWorksheetsEnabled	73
RateRoutineIndexingThreshold	73
RateTableManagementNormalizationRowLimit	73
RateTableManagementNormalizationRowThreshold	74
RatingWorksheetContainerAgeForPurging	74
Scheduler and Workflow Parameters	74
SchedulerEnabled	74
WorkflowLogDebug	74
WorkflowLogPurgeDaysOld	74
WorkflowPurgeDaysOld	75
WorkflowStatsIntervalMins	75
Search Parameters	75
ContactSearchMaxResult	75
FreeTextSearchEnabled	75
PolicySearchMaxResult	75

Security Parameters	76
EnableDownlinePermissions	76
FailedAttemptsBeforeLockout	76
LockoutPeriod	76
LoginRetryDelay	76
MaxPasswordLength	76
MinPasswordLength	76
RestrictContactPotentialMatchToPermittedItems	77
RestrictSearchesToPermittedItems	77
SessionTimeoutSecs	77
Side-by-Side Quoting Parameters	77
RenewalMaxSideBySideQuotes	77
SideBySide	77
SubmissionMaxSideBySideQuotes	78
PolicyChangeMaxSideBySideQuotes	78
User Interface Parameters	78
ActionsShortcut	78
AutoCompleteLimit	78
InputMaskPlaceholderCharacter	78
ListViewPageSizeDefault	78
MaxBrowserHistoryItems (Obsolete)	78
QuickJumpShortcut	78
UISkin	79
WizardNextShortcut	79
WizardPrevShortcut	79
WizardPrevNextButtonsVisible	79
Work Queue Parameters	79
InstrumentedWorkerInfoPurgeDaysOld	79
WorkItemCreationBatchSize	79
WorkItemPriorityMultiplierSecs	79
WorkItemRetryLimit	80
WorkQueueHistoryMaxDownload	80
WorkQueueThreadPoolMaxSize	80
WorkQueueThreadPoolMinSize	80
WorkQueueThreadsKeepAliveTime	80

Part II

The Guidewire Development Environment

3 Working with Guidewire Studio	83
What Is Guidewire Studio?	83
Using Studio with IntelliJ IDEA Ultimate Edition	84
Starting Guidewire Studio	84
Restarting Studio	84
The Studio Development Environment	85
Improving Studio Performance	86
Improving Performance of Code Compilation	86
Increasing Memory Available to Studio	86
Working with the QuickStart Development Server	87
Connecting the Development Server to a Database	88
Deploying Your Configuration Changes	88
PolicyCenter Configuration Files	89
Key Directories	89
Setting Font Display Options	90

4 PolicyCenter Studio and Gosu	93
Studio and the DCE VM.....	93
Gosu Building Blocks.....	94
Gosu Case Sensitivity.....	95
Working with Gosu in PolicyCenter Studio.....	95
Gosu Packages	95
Gosu Classes.....	96
PolicyCenter Base Configuration Classes.....	96
Class Visibility in Studio	98
Preloading Gosu Classes.....	98
Gosu Enhancements	99
The Guidewire XML Model.....	100
Script Parameters	100
Script Parameters Overview	100
Working with Script Parameters.....	101
Referencing a Script Parameter in Gosu.....	102
PolicyCenter Script Parameters.....	102

Part III

Guidewire Studio Editors

5 Using the Studio Editors	107
Editing in Guidewire Studio	107
Working in the Gosu Editor	108
Using Product Designer to Edit the Product Model	108
6 Using the Plugins Registry Editor.....	109
What Are Plugins	109
Plugin Implementation Classes.....	109
What is the Plugins Registry?.....	109
Startable Plugins	110
Working with Plugins.....	110
Creating a Plugins Registry Item	110
Adding an Implementation to a Plugins Registry Item.....	110
Setting Environment and Server Context for Plugin Implementations.....	112
Customizing Plugin Functionality	112
Working with Plugin Versions	112
7 Working with Web Services.....	115
Web Services and Guidewire Studio	115
Using the Web Service Editor	116
Defining a Web Service Collection.....	116
8 Implementing QuickJump Commands	119
What Is QuickJump?.....	119
Adding a QuickJump Navigation Command	120
Implementing QuickJumpCommandRef Commands	120
Implementing StaticNavigationCommandRef Commands.....	122
Implementing ContextualNavigationCommandRef Commands	122
Checking Permissions on QuickJump Navigation Commands	122
9 Using the Entity Names Editor.....	125
Entity Names Editor	125

Variable Table.....	126
The Entity Path Column	126
The Use Entity Names? Column.....	127
The Sort Columns	127
Gosu Text Editor.....	128
Including Data from Subentities.....	128
Entity Name Types.....	129
10 Using the Messaging Editor.....	131
Messaging Editor	131
Adding a Messaging Environment	131
Adding a Message Destination	132
Associating Event Names with a Message Destination	134
11 Using the Display Keys Editor.....	137
Display Keys Editor	137
Creating Display Keys in a Gosu Editor.....	138
Retrieving the Value of a Display Key.....	138

Part IV

Data Model Configuration

12 Working with the Data Dictionary	143
What is the Data Dictionary?	143
What Can You View in the Data Dictionary?	144
Using the Data Dictionary	144
Field Colors.....	145
Object Attributes.....	145
Entity Subtypes.....	146
Data Column and Field Types	146
Virtual Properties on Data Entities	147
13 The PolicyCenter Data Model.....	149
What is the Data Model?	149
The Data Model in Guidewire Application Architecture	150
The Base Data Model	150
Working with Dot Notation	150
Overview of Data Entities.....	151
Data Entity Metadata Files	151
Working with Data Entity Definition Files.....	154
Search for an Existing Entity Definition.....	154
Create a New Entity Definition.....	154
Extend an Existing Entity Definition	154
PolicyCenter Data Entities	155
Data Entities and the Application Database	155
PolicyCenter Database Tables.....	157
Data Objects and Scriptability	158

Base PolicyCenter Data Objects	160
Component Data Objects	160
Delegate Data Objects.....	161
Delete Entity Data Objects	164
Entity Data Objects	164
Extension Data Objects.....	169
NonPersistent Entity Data Objects	170
Subtype Data Objects	172
viewEntity Data Objects	174
viewEntityExtension Data Objects	176
Data Object Subelements	177
<array>	179
<column>	181
<componentref>	186
<edgeForeignKey>	187
<events>	190
<foreignkey>.....	191
<fulldescription>.....	194
<implementsEntity>	194
<implementsInterface>	195
<index>	195
<onetoone>	197
<remove-index>	198
<typekey>	199
14 Working with Associative Arrays	203
Overview of Associative Arrays	203
Associative Array Mapping Types	204
Scriptability and Associative Arrays.....	204
Issues with Setting Array Member Values	205
Subtype Mapping Associative Arrays	205
Working with Array Values Using Subtype Mapping	206
Typelist Mapping Associative Arrays	207
Working with Array Values Using Typelist Mapping	208
15 Modifying the Base Data Model.....	211
Planning Changes to the Base Data Model.....	211
Overview of Data Model Extension	211
Strategies for Extending the Base Data Model.....	212
What Happens If You Change the Data Model?.....	213
Naming Restrictions for Extensions	214
Defining a New Data Entity	214
Extending a Base Configuration Entity	215
Working with Attribute Overrides	216
Extending the Base Data Model: Examples	218
Creating a New Delegate Object	218
Extending a Delegate Object	220
Defining a Subtype	223
Defining a Reference Entity	224
Defining an Entity Array.....	224
Implementing a Many-to-Many Relationship Between Entity Types	226
Extending an Existing View Entity	226

Removing Objects from the Base Configuration Data Model	227
Removing a Base Extension Entity	228
Removing an Extension to a Base Object	229
Implications of Modifying the Data Model	229
Deploying Data Model Changes to the Application Server	231
16 Data Types.....	233
Overview of Data Types.....	233
Working with Data Types.....	234
Using Data Types	234
Defining a Data Type for a Property.....	235
The Data Types Configuration File	236
<...DataType>	236
Deploying Modifications to Data Types Configuration File	236
Customizing Base Configuration Data Types	237
List of Customizable Data Types	238
Working with the Medium Text Data Type (Oracle).....	239
The Data Type API.....	239
Retrieving the Data Type for a Property.....	240
Retrieving a Particular Data Type in Gosu.....	240
Retrieving a Data Type Reflectively.....	240
Using the IDataType Methods	240
Defining a New Data Type: Required Steps.....	241
Defining a New Tax Identification Number Data Type	241
Step 1: Register the Data Type	242
Step 2: Implement the IDataTypeDef Interface	242
Step 3: Implement the Data Type Aspect Handlers	243
17 The Archiving Domain Graph	247
Domain Graph Overview	247
The Domain Graph Is a Directed Acyclic Graph	248
The Domain Graph and Object Graphs in the Database.....	248
Object Ownership in the Domain Graph	248
Ownership in the Domain Graph Through Foreign Keys.....	248
Inverse Ownership in the Domain Graph	249
Ownership Through the Effective Dated Branch	249
Accessing the Domain Graph	250
Viewing the Textual Domain Graph.....	250
Viewing the Visual Domain Graph.....	250
Including Objects in the Domain Graph	251
Implementing the Correct Delegate	252
Defining Ownership Relations Between Objects	254
Domain Graph Validation.....	255
Graph Validation Errors That Prevent the Server From Starting	256
Graph Validation Warnings That Let the Server Start	256
Working with Changes to the Data Model.....	257
Working with Shared Entity Data.....	257
Eliminating Cycles From the Domain Graph	258
Circular Foreign Key References	258
Ownership Cycles	258
18 Field Validation	259
Field Validators.....	259

Field Validator Definitions	260
<FieldValidators>	261
<ValidatorDef>	261
Modifying Field Validators	263
Using <columnOverride> to Modify Field Validation	263
19 Working with Typelists	265
What is a Typelist?	266
Terms Related to Typelists	266
Typelists and Typecodes	266
Typelist Definition Files	267
Different Kinds of Typelists	268
Internal Typelists	268
Extendable Typelists	269
Custom Typelists	269
Working with Typelists in Studio	269
The Typelists Editor	269
Entering Typecodes	271
Typekey Fields	272
Removing or Retiring a Typekey	274
Removing a Typekey	275
Typelist Filters	275
Static Filters	276
Creating a Static Filter Using Categories	277
Creating a Static Filter Using Includes	278
Creating a Static Filter Using Excludes	279
Dynamic Filters	280
Creating a Dynamic Filter	281
Typecode References in Gosu	283
Mapping Typecodes to External System Codes	284

Part V

User Interface Configuration

20 Using the PCF Editor	289
Page Configuration (PCF) Editor	289
Page Canvas Overview	290
Creating a New PCF File	290
Working with Shared or Included Files	291
Understanding PCF Modes	292
Setting a PCF Mode	292
Creating New Modal PCF files	293
Page Config Menu	293
Toolbox Tab	294
Structure Tab	294
Properties Tab	295
Child Lists	296
PCF Elements	297
PCF Elements and the Properties Tab	297

Working with Elements	297
Adding an Element to the Canvas.....	298
Moving an Element on the Canvas.....	298
Changing the Type of an Element.....	299
Adding a Comment to an Element	299
Finding an Element on the Canvas	300
Viewing the Source of an Element	300
Duplicating an Element.....	300
Deleting an Element	300
Copying an Element	301
Cutting an Element	301
Pasting an Element	301
Linking Widgets	301
21 Introduction to Page Configuration.....	303
Page Configuration Files	303
Page Configuration Elements	303
What is a PCF Element?	304
Types of PCF Elements.....	305
Identifying PCF Elements in the User Interface.....	306
Getting Started Configuring Pages	309
Finding an Existing Element To Edit	309
Creating a New Standalone PCF Element	310
Modifying Style and Theme Elements	311
Changing or Adding Images.....	311
Overriding CSS.....	311
Changing Theme Colors	312
Advanced Re-Theming	312
22 Data Panels	313
Panel Overview.....	313
Detail View Panel	313
Define a Detail View.....	314
Add Columns to a Detail View	315
Format a Detail View	316
List View Panel	318
Define a List View	319
Iterate a List View Over a Data Set	321
Choose the Data Source for a List View.....	322
23 Navigation	325
Navigation Overview	325
Tab Bars	326
Configure the Default Tab Bar	326
Specify Which Tab Bar to Display	326
Define a Tab Bar	327
Tabs.....	327
Define a Tab	327
Define a Drop-down Menu on a Tab	327
24 Configuring Search Functionality	329
Search Overview	329

Database Search Configuration	331
PolicyCenter Database Search Functionality	331
Configuring PolicyCenter Database Search	332
Working with Database Search Criteria in XML	335
Working with Database Search Criteria in Gosu	337
Free-text Search Configuration.....	341
Overview of Free-text Search	342
Free-text Search System Architecture.....	342
Enabling Free-text Search in PolicyCenter	346
Configuring the Solr Extension for Integration with PolicyCenter	347
Configuring Free-text Search for Indexing and Searching	350
Configuring the Free-text Batch Load Command	350
Configuring the Basic Search Screen for Free-text Search	351
Modifying Free-text Search for Additional Fields	352
25 Configuring Special Page Functions	357
Adding Print Capabilities	357
Overview of the Print Functionality	357
List View Printing.....	358
Linking to a Specific Page: Using an EntryPoint PCF.....	359
Entry Points.....	359
Creating a Forwarding EntryPoint PCF	361
Linking to a Specific Page: Using an ExitPoint PCF	361
Creating an ExitPoint PCF	361
Part VI	
Workflow and Activity Configuration	
26 Using the Workflow Editor	367
Workflow in Guidewire PolicyCenter	367
Workflow in Guidewire Studio.....	368
Understanding Workflow Steps	369
Using the Workflow Right-Click Menu	370
Using Search with Workflow	370
27 Guidewire Workflow	373
Understanding Workflow	373
Workflow Instances	374
Work Items	375
Workflow Process Format.....	375
Workflow Step Summary	376
Workflow Gosu.....	376
Workflow Versioning	377
Workflow Localization	378
Workflow Structural Elements	378
<Context>	379
<Start>.....	379
<Finish>	379
Common Step Elements	379
Enter and Exit Scripts	380
Asserts.....	380
Events	380
Notifications	381
Branch IDs	381

Basic Workflow Steps	381
AutoStep	381
MessageStep	382
ActivityStep	383
ManualStep	384
Outcome	385
Step Branches	386
Working with Branch IDs.....	387
GO.....	387
TRIGGER.....	388
TIMEOUT.....	390
Creating New Workflows.....	391
Cloning an Existing Workflow.....	391
Extending an Existing Workflow	391
Extending a Workflow: A Simple Example	392
Instantiating a Workflow	395
A Simple Example of Instantiation.....	395
The Workflow Engine	397
Distributed Execution	397
Synchronicity, Transactions, and Errors.....	398
Workflow Subflows	400
Workflow Administration.....	401
Workflow Debugging, Logging, and Testing.....	402
28 Defining Activity Patterns	405
What is an Activity Pattern?.....	405
Pattern Types and Categories	406
Activity Pattern Types	406
Categorizing Activity Patterns	406
Using Activity Patterns in Gosu	407
Calculating Activity Due Dates	407
Target Due Dates (Deadlines).....	408
Escalation Dates	408
Configuring Activity Patterns.....	408
Using Activity Patterns with Documents and Emails.....	410
Localizing Activity Patterns	411
Part VII	
Testing Gosu Code	
29 Testing and Debugging Your Configuration	415
Testing PolicyCenter With Guidewire Studio	415
Running PolicyCenter Without Debugging	416
Debugging PolicyCenter Within Studio	416
Debugging a PolicyCenter Server That Is Running Outside of Studio.....	416
The Studio Debugger	418
Setting Breakpoints.....	418
Stepping Through Code	419
Viewing Current Values	420
Enabling the Viewing of Entities While Debugging.....	420
Viewing Variables.....	420
Defining a Watch List.....	420
Resuming Execution.....	421

Using the Gosu Scratchpad	421
Testing a Gosu Expression	421
Suggestions for Testing Rules	422
30 Using GUnit	423
The TestBase Class	423
Overriding TestBase Methods	424
Configuring the Server Environment	424
Configuring the Test Environment	426
Configuration Parameters	427
Creating a GUnit Test Class	428
Using Entity Builders to Create Test Data	430
Creating an Entity Builder	431
Entity Builder Examples	433
Creating New Builders	435

Part VIII Guidewire PolicyCenter Configuration

31 PolicyCenter Configuration Guidelines	443
Guidelines for Modularizing Line-of-business Code	443
32 Configuring Policy Archiving	445
Archiving and the Domain Graph	445
The Root Info Entity	447
Archiving Objects Example	447
Archiving in Guidewire PolicyCenter	447
Archiving and Encryption	448
Selecting Policy Terms for Archive Eligibility	448
Configuring Archive Eligibility	448
Batch Processes and Archiving	449
Batch Processes to Run Prior to Archive Policy Term Batch Process	449
Monitoring Archiving Activity	450
Configuring Archiving	450
Data Model for Archiving	450
Archiving-related Configuration Parameters	452
Gosu Code and Archiving	453
Archive Work Queue	454
Archiving Plugins	454
Archiving Test Tool	455
Flushing Other Work Queues	455
Archive Term by Job Number	455
Archive Policy Term by Policy Number and Term Number	456
Running the Archive Policy Terms Batch Process from the Tool	456
33 Configuring Quote Purging	457
Quote Purging Configuration Overview	457
Quote Purging Object Model	458
Objects that Get Purged or Pruned	459
Quote Purging Batch Processes	461
Enabling Quote Purging Batch Processes	461
Batch Processes to Run Prior to Running the Purge Batch Process	462
Purge Batch Process	462
Purge Orphaned Policy Periods Batch Process	466
Using Events to Notify External System of Purged or Pruned Entities	466

Purging Test Tool	466
Flushing Other Work Queues with the Purging Test Tool	466
Purging or Pruning a Job with the Purging Test Tool	467
Pruning Policy Periods with the Purging Test Tool	468
Running the Purge Batch Process from the Purging Test Tool	468
34 Configuring Underwriting Authority	469
Overview of Configuring Underwriting Authority	469
Raising Underwriting Issues	470
Issues Blocking Progress	471
Implementing Underwriting Authority	471
Defining an Underwriting Issue	473
Creating Underwriting Issues	474
Adding a New Checking Set	475
Adding a New Value Comparator	476
Adding a New Value Formatter	477
Configuring Authority Grants	478
Underwriting Authority Profile Object Model	478
Displaying Authority Grants in the User Interface	479
Configuring Underwriting Issues	479
Underwriting Issue Object Model	480
Checking Sets and Evaluators	481
Removing Orphaned Issues	482
Blocking Points	482
Configuring Underwriting Referral Reasons	483
Job Interactions with Underwriting Issues	483
Issue Keys	487
Configuring Underwriting Issue History	487
Comparing Issue Values	488
Underwriting Issue Type System Table	489
Issues in the Underwriting Issue Type System Table	489
Default Approval Values in Underwriting Issue Type System Table	493
Additional Information About the Underwriting Issue Type System Table	494
Configuring Approvals	495
Rejecting an Issue	495
Reopening an Issue	495
Approvals for Auto-approvable Issues	495
Approval Expiration	496
Special Approval Permission	496
Passing Approval Requests to Underwriters	496
Handling Underwriting Issues in Policy Revisions	497
Handling Underwriting Issues in Out-of-sequence Policy Changes	497
Handling Underwriting Issues in Preempted Jobs	498
35 Configuring the Account Holder Info Screen	499
Ways to Configure the Account Holder Info Screen	500
Consolidate Account Information	500
Cross-application Alerts in a Single Location	501
Links to Referenced Objects	501
Controlled Access to the Account Holder Info Screen	502
Customizing the Screen Data	502
Performance	502

Configuration Files for the Account Holder Info Screen	503
Entities and the Account Holder Info Screen	503
Permissions for the Account Holder Info Screen	503
Display Keys for the Account Holder Info Screen	503
PCF Files for the Account Holder Info Screen	503
Gosu Files for the Account Holder Info Screen	503
36 Configuring Policy Data Spreadsheet Import/Export	505
Changing the Spreadsheet Protection Password	505
Configuring Spreadsheet Import/Export in Commercial Property	506
Adding Spreadsheet Import/Export to Other Entities	506
Step 1: Create an XML File Describing Columns to Import and Export	506
Step 2: Define Column Data Resolvers	508
Step 3: Create an Import Strategy for Each New Data Column Resolver	509
Step 4: Modify the ImportStrategyRegistry Class	510
37 Configuring Earned Premium	511
How PolicyCenter Calculates Earned Premium	511
Methods that Calculate Earned Premium	512
Earned Premium	512
Earned Premium As of Date	512
PCF Files that Display Earned Premium	513
38 Configuring the Team Tab	515
Configuring the Team Screens Batch Process for Team Statistics	515
Scheduling the Team Screens Batch Process	516
Setting the Window Size for Team Statistics	516
How PolicyCenter Calculates Reporting Categories	517
Activities with a One Day Window Example	517
Submissions with a This Week Window Example	517
Renewals with a This Month Window Example	518
Other Work Orders	518
Setting the Maximum Number of Rows on the Team Screens	518
39 Configuring Rating Management	521
Rating Management Data Model	521
Rate Table Data Model	522
Rate Routine Data Model	522
Improving Rate Table Performance	523
Covering Index to Improve Performance	524
Parameters and Properties for Rating Management	524
Minimum Rating Level Parameter	524
Rate Table Normalization Configuration Parameters	525
Logging Properties for Rating Management	525
Third Party Libraries for Rating Management	526
POI Library	526
User Authority and Permissions for Rating Management	526
Rating Management Permissions	526
Rating Management Roles	526
Assignment of Permission to Roles	527
Custom Physical Tables for Rating Management	527
Configuring a New Custom Physical Table	527

Configuring Value Providers	528
Typelist Value Provider	528
PolicyCenter Product Model Value Providers	528
Reference Factor Value Provider	529
Creating a New Value Provider	530
Configuring Matching Rule Operations	530
Configuring a New Match Operation	530
Configuring Rate Routines	533
Configuring the Rating Engine to Execute the Rate Routine	533
Adding a New Rate Routine Function	534
Configuring Rounding Operators	535
Configuring Prefixes that Identify Types in Rate Routine Steps	535
Configuring Variant Identifiers for a Rate Routine	536
Configuring the Rate Routine Plugin	538
Using Gosu to Get the Rate Factor from the Rate Table	538
Configuring New Parameters in Parameter Sets	540
Adding a New Parameter to the Rating Engine	542
Configuring New Wrappers for Parameter Sets	543
Configuring Rating Worksheets	543
Configuring Extract and Purge Rating Worksheets	544
Logging Rate Routine Functions in a Rating Worksheet	546
Configuring Impact Testing	546
Enabling Impact Testing for a Line of Business	547
Rating Renewals as Renewal Jobs in Impact Testing	547
Modifying the Functionality of Impact Testing	548
Work Queues for Impact Testing	548
Configuring the Impact Testing Plugin	548
Rating Management Plugins and Interfaces	548
Rate Routine Plugin	549
Impact Testing Plugin	553
Adding Line-specific Cost Methods Using Cost Adapters	555
40 Configuring Reinsurance Management	557
Reinsurance Management Object Model	557
Reinsurance Program Object Model	557
Reinsurance Agreement Object Model	559
Reinsurance Coverage Group Object Model	561
Policy Period Reinsurance Object Model	563
Reinsurance Management Permissions	563
Configuring Reinsurance Coverage Groups	564
Configuring the Reinsurance Coverage Group on Individual Coverages	564
Disabling Reinsurance Management for a Line of Business	564
Reinsurance Management Underwriting Issues	564
Reinsurance Net Retention Underwriting Issue	564
Implementing Gosu Methods for Reinsurance Management	565
Gosu Methods for Calculating the Total Insured Value	565
41 Handling High Volume Quote Requests	567
High Volume Quotes Overview	568
Generating the Quote	568
Viewing the Quote	568
Moving the Quote into the System of Record	568
Other Considerations for Handling High Volume Quotes	569
High Volume Quotes Implementation Overview	569

Quoting Processor Class	571
Logging	571
Quoting Data Plugin	571
Get Account	571
Send Quoting Data	572
42 Configuring Multicurrency	573
Configuring PolicyCenter for a Single Currency	573
Creating Multicurrency Policy Lines	574
Adding Coverage Currencies to a Policy Line	574
Changing the Settlement Currency	576
Configuring the Coverage Currency	576
Coverage Currency in Accounts	576
Coverage Currency in Policy Periods	576
Coverage Currency in Coverages and Clauses	577
Coverage Currency in Coverables	577
Configuring the Settlement Currency	577
Settlement Currency in Contacts	577
Settlement Currency in Accounts	578
Settlement Currency in Policy Periods	578
Configuring Multicurrency and Reinsurance	579
Configuring Multicurrency and Rating	580
Costs and Multicurrency	580
Configuring Underwriting Authority and Multicurrency	582
Configuring Underwriting Issues and Multicurrency	582
Configuring Authority Profiles and Multicurrency	583
Implementing an Exchange Rate Service	583
The Exchange Rate Service Plugin Interface	583
Enabling Multicurrency Integration	584
Set up Currency-Specific Plans and Authority Limits	584

Part IX

Guidewire PolicyCenter Job Configuration

43 Configuring Jobs	589
Common Ways to Configure Jobs	589
Wizards for Jobs	590
Gosu Classes for Jobs	590
Rule Sets	591
Workflows	591
System Configuration Parameters for Jobs	591
Configuring Specific Aspects of a Job	591
Changing Jobs to Expired Status	592
Configuring Job History Events	592
Multiple Revision Jobs and the Job Selected Branch Property	595
Selecting the Underwriting Company through Segmentation	595
44 Configuring Submissions	597
Configuring Submissions Overview	597
Submission Process Gosu Class	598
Submission Enhancements	599
Submission Integration	599
Submission Web Services	599
Submission Plugins	599
Expiring Submissions	600

Submission Configuration Examples	601
Configuring the Copy Submission Feature.....	601
45 Configuring Issuance	603
Configuring Issuance Overview	603
Modifying the Issuance Process	604
Issuance Integration	604
Issuance Web Services and Plugins	604
46 Configuring Renewals	605
Configuring Renewals Overview	605
Renewal Process Gosu Class and Enhancements	606
Renewal Enhancement	607
Renewal Workflows	607
Renewal Rule Sets	607
Renewal Integration	608
Renewal Web Services and Plugins	608
Renewal Events	609
Configuring Batch Process Renewals.....	610
Renewal Plugin Calculation	611
Renewal Lead Time in Notification Config System Table.....	611
Example Lead Time by Line of Business	612
Example Lead Time by Time of Year	612
Delay for a Conflicting Job	612
Configuring Explanations in Pre-renewal Directions.....	613
47 Configuring Cancellations	615
Configuring Cancellations Overview	615
Cancellation Gosu Classes	616
Complete Cancellation Workflow	617
Cancellation Integration	617
Cancellation Web Services and Plugins	617
Events	618
Calculating the Cancellation Effective Date.....	619
Configuring the Premium Calculation Method	620
48 Configuring Policy Change	621
Configuring Policy Change Overview	621
Policy Change Process Class	622
Modifying the Policy Change Process Class	623
Policy Change Integration	623
Policy Change Web Services and Plugins	623
Policy Change Events	624
49 Configuring Reinstatement	625
Configuring Reinstatement Overview	625
Reinstatement Process Gosu Class	626
Reinstatement Integration	626
Reinstatement Web Services and Plugins.....	626
Events	627
50 Configuring Rewrite	629
Configuring Rewrite Overview	629
Rewrite Process Gosu Class	630
Rewrite Integration	630
Events	630

51 Configuring Rewrite New Account	631
Configuring the Rewrite New Account Job Overview.....	631
Rewrite New Account Job	631
Rewrite New Account Object Model	632
Rewrite New Account History Events	633
52 Configuring Premium Audit.....	635
Configuring Premium Audit Overview	635
Audit Schedules	636
Audit Gosu Classes	637
Final Audit Integration	638
Integration with BillingCenter	638
Audit Web Services and Plugins.....	638
Events	639
53 Configuring Side-by-side Quoting.....	641
Changing the Maximum Number of Versions in Side-by-side Quoting	641
Marking a Job as Side-by-side	642
Side-by-side Quoting Object Model.....	642
Job Entity	642
Base Data Entities.....	642
Configuring the Side-by-side Quoting Screen	642
Posting Changes to the Server	643
Disabling Post on Enter.....	643
Configuring Initial Behavior of the Side-by-side Quoting Screen	643
Copying Base Data for Side-by-side Quoting	644
High-level View of Base Data Copy.....	644
Gosu Methods for Base Data Copy.....	645
Excluding Side-by-side Data from Base Data	647
Side-by-side Data that Base Data Copy Must Exclude.....	647
Side-by-side Data that Base Data Copy Can Exclude	648
Side-by-side Data in Personal Auto Excluded from Base Data Copy	650
Exclude Entities Reachable Through Two Paths	650
Configuring Data Excluded from Base Data	653
Configuring Quote All to Ignore Validation Warnings	654

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://\$ERVERNAME/a.html</code> .

Support

For assistance with this release, go to the Guidewire Resource Portal at <http://guidewire.custhelp.com>.

part I

PolicyCenter Configuration Basics

Overview of PolicyCenter Configuration

This topic provides some basic information, particularly about the Guidewire PolicyCenter data model and system environment. Guidewire recommends that before you undertake any configuration changes, that you thoroughly understand this information.

This topic includes:

- “What You Can Configure” on page 29
- “How You Configure PolicyCenter” on page 30
- “Types of Application Environments” on page 31
- “Deploying Configuration Files” on page 31
- “Regenerating the Data Dictionary and Security Dictionary” on page 32
- “Managing Configuration Changes” on page 34

What You Can Configure

Application configuration files determine virtually every aspect of the PolicyCenter application. For example, you can make the following changes by using XML configuration files and simple Gosu:

Extend the Data Model

You can add fields to existing entities handled by the application, or create wholly new entities to support a wide array of different business requirements. You can:

- Add a field such as a column, typekey, array or foreign key to an extendable entity. For example, you can add an `IM Handle` field to contact information.
- Create a subtype of an existing non-final entity. For example, you can create an `Inspector` object as a subtype of `Person`.

- Create a new entity to model an object not supported in the base configuration product. For example, you can create an entity to archive digital recordings of customer phone calls.

Change or Augment the PolicyCenter Application

You can extend the functionality of the application:

- Build new views of policies and other associated objects.
- Create or edit validators on fields in the application.
- Configure a new line of business.

Modify the PolicyCenter Interface

You can configure the text labels, colors, fonts, and images that comprise the visual appearance of the PolicyCenter interface. You can also add new screens and modify the fields and behavior of existing screens.

Implement Security Policies

You can customize permissions and security in XML and then apply these permissions at application runtime.

Create Business Rules and Processes

You can create customized business-specific application rules and Gosu code. For example, you can create business rules that perform specialized tasks for validation and work assignment.

Designate Activity Patterns

You can group work activities and assign to agents and underwriters.

Integrate with External Systems

You can integrate PolicyCenter data with external applications.

Define Application Parameters

You can configure basic application settings such as database connections, clustering, and other application settings that do not change during server runtime.

Define Workflows

You can create new workflows, create new workflow types, and attach entities to workflows as context entities. You can also set new instances of a workflow to start within Gosu business rules.

How You Configure PolicyCenter

Guidewire provides a configuration tool, Guidewire Studio, that you use to edit files stored in the development environment. (Guidewire also calls the development environment the *configuration* environment.) You then deploy the configuration files by building a .war or .ear file and moving it to the application (production) server.

Guidewire Studio provides graphical editors for most of the configuration files. A few configuration files you must manually edit (again, through Studio).

See also

- For information on Guidewire Studio, see the “Using the Studio Editors” on page 107.

Types of Application Environments

Configuring the application requires an installed development instance of the application (often called a *configuration environment*). You use Guidewire Studio to edit the configuration files. Then, you create a .war or .ear file and copy it to the *production* server. This section describes some of the particular requirements of these two environments:

- The Development Environment
- The Production Environment

The Development Environment

The development environment for PolicyCenter has the following characteristics:

- It includes an embedded development QuickStart server and database for rapid development and deployment.
- It includes a repository for the source code of your customized application files. Guidewire expects you to check your source code into a supported Software Configuration Management—SCM—system.
- It includes directories for the base configuration application files and your modifications of them.
- It provides command line tools for creating the deployment packages. (These are new, customized, versions of the server application files that you use in a production environment.)

Guidewire provides a development environment (Guidewire Studio) that is separate from the production environment. Guidewire Studio is a stand-alone development application that runs independently of Guidewire PolicyCenter. You use Studio to build and test application customization in a development or test mode before deploying your changes to a production server. (You can for example, modify a PCF file or add a new workflow.)

It is important to understand that any changes that you make to application files through Studio do not automatically propagate into your production environment. You must specifically build a .war or .ear file and deploy it to a production server for the changes to take effect. Studio and the production application server—by design—do not share the same configuration file system.

The Production Environment

The deployed production application server for PolicyCenter has the following characteristics:

- It runs as an application within an application server.
- It handles web clients, or SOAP message requests, for policy information or services.
- It generates the web pages for browser-based client access to PolicyCenter.

Deploying Configuration Files

There is a vast difference in how you deploy modified configuration files in a development environment as opposed to a production environment. The following sections describes these differences:

- Deploying Changes in a Development Environment
- Deploying Changes to the Production Server

Deploying Changes in a Development Environment

In the base configuration, Guidewire provides an embedded application server in the development environment. This, by design, shares its file structure with the Studio application configuration files. Thus:

- If you modify a file, in many cases, you do not need to deploy the changed configuration file. The development server reflects the changes automatically. For example, if you add a new typelist, Studio recognizes this change.

- If you modify certain resource files, you must stop and restart Studio for the change to become effective. For example, if you add a new workflow type, then you must stop and restart Studio before a Gosu class that you create recognizes the workflow.
- If you modify the base configuration data model files, you must stop and restart the development server to force a database upgrade.

It is possible to use a different development environment and database other than that provided by Guidewire in the base configuration. If you do so, then deployment of modified configuration files can require additional work. For details on implementing a different development environment, see “Selecting an Installation Scenario” on page 10 in the *Installation Guide*.

Deploying Changes to the Production Server

To deploy configuration changes to the PolicyCenter production application server, you must do the following:

- Create an application .war (or .ear) file with your configuration changes in the development environment.
- Shut down the production server.
- Remove the old PolicyCenter instance on the production application server.
- Deploy the .war (or .ear) file to the production application server.
- Restart the production application server.

In the following procedure, notice whether you need to do a task on the *development* or *production* server.

To deploy a .war (.ear) file

1. After making configuration changes in your development environment, run one of the `build-*` commands from your *configuration* PolicyCenter `bin` directory. For example:
`gwpc build-tomcat-war-dbc`
2. Shut down the *production* application server.
3. Delete the existing web application folder in the *production* server installation. For example (for Tomcat), delete the following folder:
`PolicyCenter/webapps/pc`
Also, delete the existing .war (or .ear) file on the production server. In any case, moving a new copy to the production server overwrites the existing file.
4. Navigate to your *configuration* installation `dist` directory (for example, `PolicyCenter/dist`). The `build` script places the new `pc.war` or `pc.ear` file in this directory.
5. Copy the newly created `pc.war` file to the *production* `webapps` folder (for Tomcat). If using WebSphere or WebLogic, use the administrative console to deploy the `pc.ear` file.
6. Restart the *production* application server.
7. During a server start, if the application recognizes changes to the data model, then it mandates that a database upgrade be run. The server runs the database upgrade automatically.

Regenerating the Data Dictionary and Security Dictionary

If you change the metadata, for example by extending base entities, it is important that you regenerate the *Data Dictionary* and *Security Dictionary* to reflect those changes. In this way, other people who use the dictionaries can see these changes. You can generate the *Data Dictionary* and the *Security Dictionary* in HTML or XML format.

See also

- To understand the *Data Dictionary* and the information it includes, see “Working with the Data Dictionary” on page 143.
- To understand the *Security Dictionary* and the information it includes, see “Security Dictionary” on page 665 in the *Application Guide*.

Generating the Data and Security Dictionaries in HTML Format

To generate the *PolicyCenter Data Dictionary* and *PolicyCenter Security Dictionary* in HTML format, run the following command from the *PolicyCenter/bin* directory:

```
gwpc regen-dictionary
```

This command generates HTML files for the dictionaries in the following locations:

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

To view a dictionary, open its `index.html` file from the listed locations.

Generating the Data and Security Dictionaries in XML Format

You can generate the *Data Dictionary* and *Security Dictionary* in XML format, with associated XSD files. Use the generated XML and XSD files to import the *Data Dictionary* and *Security Dictionary* into third-party database design tools.

To generate the *Data Dictionary* and *Security Dictionary* in XML format, run the following command from the *PolicyCenter/bin* directory:

```
gwpc regen-dictionary -DoutputFormat=xml
```

This command generates the following XML and XSD files for the dictionaries:

```
PolicyCenter/build/dictionary/data/entityModel.xml  
PolicyCenter/build/dictionary/data/entityModel.xsd  
  
PolicyCenter/build/dictionary/security/securityDictionary.xml  
PolicyCenter/build/dictionary/security/securityDictionary.xsd
```

The parameter that specifies the output format has two allowed values.

```
regen-dictionary -DoutputFormat={html|xml}
```

If you specify `-DoutputFormat=html` or you omit the `-DoutputFormat` parameter, the `regen-dictionary` command generates HTML versions of the *Data Dictionary* and the *Security Dictionary*.

Generating the Dictionaries as You Generate a .war or .ear File

You can generate the *Data Dictionary* and the *Security Dictionary* in HTML format as you generate the Guidewire application `.war` (`.ear`) file. To do so, use one of the `build-*` commands. For example:

```
gwpc build-tomcat-war-dbcp -Dconfig.war.dictionary=true
```

See also

- For information on the Guidewire command line tools for use in a development environment, see “Commands Reference” on page 117 in the *Installation Guide*.

Aspects of Regenerating the Security Dictionary

Guidewire PolicyCenter stores the role information used by the *Security Dictionary* in the base configuration in the following file:

```
PolicyCenter/modules/configuration/config/import/gen/roleprivileges.csv
```

PolicyCenter does not write this file information to the database.

If you make changes to roles using the PolicyCenter interface, PolicyCenter does write these role changes to the database.

PolicyCenter does not base the *Security Dictionary* on the actual roles that you have in your database. Instead, PolicyCenter bases the *Security Dictionary* on the `roleprivileges.csv` file.

IMPORTANT It is possible for the *Security Dictionary* to become out of synchronization with the actual roles in your database. Guidewire PolicyCenter bases the *Security Dictionary* on file `roleprivileges.csv` only.

Managing Configuration Changes

To track, troubleshoot and replicate the configuration changes that you make, Guidewire strongly recommends that you use a Software Configuration Management (SCM) to manage the application source code.

Application Configuration Parameters

This topic covers the PolicyCenter configuration parameters, which are static values that you use to control various aspects of system operation. For example, you can control business calendar settings, cache management, and user interface behavior through the use of configuration parameters, along with much more.

This topic includes:

- “Working with Configuration Parameters” on page 36
- “Archive Parameters” on page 38
- “Assignment Parameters” on page 39
- “Batch Process Parameters” on page 39
- “Business Calendar Parameters” on page 39
- “Cache Parameters” on page 41
- “Clustering Parameters” on page 44
- “Database Parameters” on page 47
- “Desktop and Team Parameters” on page 48
- “Document Creation and Document Management Parameters” on page 50
- “Domain Graph Parameters” on page 51
- “Environment Parameters” on page 52
- “Financial Parameters” on page 56
- “Geocoding Feature Parameters” on page 56
- “Globalization Parameters” on page 57
- “Integration Parameters” on page 60
- “Job Expiration Parameters” on page 62
- “Lookup Table Parameters” on page 64
- “Miscellaneous Job-Related Parameters” on page 64
- “Miscellaneous Parameters” on page 66
- “PDF Print Settings Parameters” on page 68

- “Product Model” on page 71
- “Quote Purging Configuration Parameters” on page 71
- “Rating Management Parameters” on page 73
- “Scheduler and Workflow Parameters” on page 74
- “Search Parameters” on page 75
- “Security Parameters” on page 76
- “Side-by-Side Quoting Parameters” on page 77
- “User Interface Parameters” on page 78
- “Work Queue Parameters” on page 79

Working with Configuration Parameters

You set application configuration parameters in the file `config.xml`. You can find this file in the Guidewire Studio Resources panel in the Other Resources folder. If you open this file for editing, Studio makes a copy of the read-only base configuration file and places the editable copy in the following directory:

`PolicyCenter/modules/configuration/config`

You do not ever need to touch this file directly outside of Studio other than to check it into your source control system. Because Guidewire PolicyCenter maintains several copies of this file in different locations, always use Studio to modify configuration files to let Studio manage the various copies for you.

WARNING Do not modify any files other than those in the `/modules/configuration` directory. Specifically, do not modify files in the `/modules/pc` directory. Any modification of files in this directory can cause damage to the PolicyCenter application sufficient to prevent the application from starting.

This file generally contains entries in the following format:

```
<param name="param_name" value="param_value" />
```

Each entry sets the parameter named `param_name` to the value specified by `param_value`.

The standard `config.xml` file contains all available parameters. To set a parameter, edit the file, locate the parameter, and change its value. PolicyCenter configuration parameters are case-insensitive. If a parameter does not appear in the file, Guidewire PolicyCenter uses the default value, if the parameter has one.

Adding Custom Parameters to PolicyCenter

You cannot add new or additional configuration parameters to `config.xml`. Guidewire does not support any attempt to do so. If you want to add custom parameters to your configuration of PolicyCenter, consider defining script parameters. You can access the values of script parameters in Gosu code at runtime. For more information, see “Script Parameters” on page 100.

Accessing Configuration Parameters in Gosu

To access a configuration parameter in Gosu code, use the following syntax:

```
gw.api.system.PLConfigParameters  
gw.api.system.PCConfigParameters
```

For example:

```
var businessDayEnd = gw.api.system.PLConfigParameters.BusinessDayEnd.Value  
var forceUpgrade = gw.api.system.PLConfigParameters.ForceUpgrade.Value
```

Configuration Parameter Attributes

The configuration parameters in `config.xml` use the following attributes:

- Required
- Set for Environment
- Development Environment Only
- Permanent

Required

Guidewire designates certain configuration parameters as `required`. This indicates that you must supply a value for that parameter. The discussion of configuration parameters indicates this by adding *Required: Yes* to the parameter description.

Set for Environment

Guidewire designates certain configuration parameters as `localok`. This indicates that it is possible for this value to vary on different servers in the same environment. For example, you can set `ClusterProtocolStackOption1` to a value that is very specific to a given host, with `xxx.xxx.xxx.xxx` being the host IP address:

```
;bind_addr=xxx.xxx.xxx.xxx
```

The discussion of configuration parameters indicates this by adding *Set for Environment: Yes* to the parameter description.

Guidewire prints a warning message if you attempt to add a configuration parameter that Guidewire does not designate as `localok` to a local configuration file. PolicyCenter prints the warning to the configuration log at start of the application server. For example:

```
WARN Illegal parameter specified in a local config file (will be ignored): XXXXXXXX
```

Note: For information on server environments in Guidewire PolicyCenter, see “Defining the Application Server Environment” on page 14 in the *System Administration Guide*.

Development Environment Only

Guidewire designates certain configuration parameters as `devonly`. This indicates that Guidewire permits this configuration parameters to be active in a development environment only. Thus, a production server ignores any configuration parameter for which `devonly` is set to `true`. The discussion of configuration parameters indicates this by adding *Development Environment Only: Yes* to the parameter description.

Permanent

Guidewire specifies several configuration parameter values as `permanent`. This indicates that after you set such a parameter and start the production application server that you cannot change the value thereafter. This applies, for example, to the `DefaultApplicationLocale` configuration parameter. If you set this value on a production server and then start the server, you are unable to change the value thereafter.

Guidewire stores these values in the database and checks the value at server start up. If an application server value does not match a database value, PolicyCenter throws an error.

Adding Custom MIME Types

Adding a new MIME type (MIME stands for Multipurpose Internet Mail Extensions), such that PolicyCenter recognizes it and so that it flows smoothly through the application, requires the following steps:

1. Add the MIME type to the configuration of the application server (if required). This depends on the details of the application servers configuration.

For example, Tomcat stores MIME type information in the `web.xml` configuration file, in a series of `<mime-mapping>` tags. Verify that the MIME type you need already exists (correctly) in this list, or add it.

2. Add the new MIME type to the <mimetypemapping> section of config.xml. You need to add the following items:

- The name of the MIME type, which is the same as the identifying string ("text/plain", for example).
- The file extensions to be used for the MIME type. If more than one apply, separate them with a "|". PolicyCenter uses this information to map from MIME type to file extension and file extension to MIME type. If mapping from type to extension, PolicyCenter uses the first extension in the list. If mapping from extension to type, PolicyCenter uses the first <mimetype> entry containing that extension.
- The image, in the tomcat/webapps/pc/resources/images directory (or equivalent), to use for documents of this MIME type.
- Human-readable description of the MIME type, for logging and documentation purposes.

Archive Parameters

Guidewire provides the following configuration parameters in the config.xml file related to archiving. Archiving is the process of moving a closed claim and associated data from the active PolicyCenter database to a document storage area. You can still search for and retrieve archived claims. But, while archived, these claims occupy less space in the active database.

For information on editing config.xml and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

See also

- “More Information on Archiving” on page 325 in the *Application Guide* for a list of topics related to archiving.

IMPORTANT Guidewire strongly recommends that you contact Customer Support before implementing archiving.

ArchiveEnabled

Whether archiving is enabled (set to true) or disabled (set to false). Default is false. For archiving to work, you must set ArchiveEnabled to true and configure an archive database.

This parameter also controls the creation of indexes on the ArchivePartition column. If you set the value of this parameter to true, PolicyCenter creates a non-unique index on the ArchivePartition column for Extractable entities.

In PolicyCenter, this parameter controls whether or not the domain graph is started.

WARNING If you set ArchiveEnabled to true, the server refuses to start if you subsequently set the parameter to false.

Default: False

Required: Yes

Permanent: Yes

ArchiveDaysRetrievedBeforeArchive

Minimum number of days that must pass after a term is retrieved before PolicyCenter can archive it again.

Default: 90

ArchiveDefaultRecheckDays

Minimum number of days that must pass before PolicyCenter reconsiders a policy term for archiving after failing a generic archiving check.

Default: 30

ArchivePolicyTermDays

Minimum number of days that must pass before the policy periods associated with a given policy term can be archived.

Default: 366

ArchiveRecentJobCompletionDays

Minimum number of days that must pass after a job is closed before archiving the associated policy term. Jobs are associated with policy periods.

Default: 90

Assignment Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to assignment.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

AssignmentQueuesEnabled

Whether to display the PolicyCenter interface portions of the assignment queue mechanism. If you turn this on, you cannot turn it off again while working with the same database.

Default: false

Batch Process Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to batch processing.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

BatchProcessHistoryPurgeDaysOld

Number of days to retain batch process history before PolicyCenter deletes it.

Default: 45

Business Calendar Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to defining a business calendar.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

BusinessDayDemarcation

The point in time at which a business day begins.

Default: 12:00 AM

Set For Environment: Yes

BusinessDayEnd

Indicates the point in time at which the business day ends.

Default: 5:00 PM

Set For Environment: Yes

BusinessDayStart

Indicates the point in time at which the business day starts.

Default: 8:00 AM

Set For Environment: Yes

BusinessWeekEnd

The day of the week considered to be the end of the business week.

Default: Friday

Set For Environment: Yes

HolidayList (Obsolete)

This parameter is obsolete. Do not use it. Formerly, you would use this to generate a comma-delimited list of dates to consider as holidays. Instead, use the **Administration** screen within Guidewire PolicyCenter to manage the official designation of holidays. Guidewire retains this configuration parameter to facilitate upgrade from older versions of PolicyCenter.

IsFridayBusinessDay

Indicates whether Friday is a business day.

Default: True

Set for Environment: Yes

IsMondayBusinessDay

Indicates whether Monday is a business day.

Default: True

Set for Environment: Yes

IsSaturdayBusinessDay

Indicates whether Saturday is a business day.

Default: False

Set for Environment: Yes

IsSundayBusinessDay

Indicates whether Sunday is a business day.

Default: False

Set for Environment: Yes

IsThursdayBusinessDay

Indicates whether Thursday is a business day.

Default: True

Set for Environment: Yes

IsTuesdayBusinessDay

Indicates whether Tuesday is a business day.

Default: True

Set for Environment: Yes

IsWednesdayBusinessDay

Indicates whether Wednesday is a business day.

Default: True

Set for Environment: Yes

MaxAllowedDate

The latest date allowed to be used.

Default: 2200-12-31

Set For Environment: Yes

MinAllowedDate

The earliest date allowed to be used.

Default: 1800-01-01

Set For Environment: Yes

Cache Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to the application cache.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

See also

- “Application Server Caching” on page 65 in the *System Administration Guide*

ExchangeRatesCacheRefreshIntervalSecs

The time between refreshes of the ExchangeRateSet cache, in seconds. This integer value must be zero (0) or greater. See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 600

GlobalCacheActiveTimeMinutes

Time period (in minutes) in which PolicyCenter considers cached items as *active*, meaning that Guidewire recommends that the cache give higher priority to preserve these items. You can think of this as the period during which PolicyCenter is using one or more items very heavily. For example, this can be the length of time that a user stays on a page. Make this value less than the reaping time (*GlobalCacheReapingTimeMinutes*). See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 5

Minimum: 1

Maximum: 15

Set for Environment: Yes

GlobalCacheDetailedStats

Configuration parameter `GlobalCacheDetailedStats` determines whether to collect detailed statistics for the global cache. Detailed statistics are data that PolicyCenter collects to explain why items are evicted from the cache. PolicyCenter collects basic statistics, such as the miss ratio, regardless of the value of this parameter.

In the base configuration, Guidewire sets the value of the `GlobalCacheDetailedStats` parameter to `false`. Set the parameter to `true` to help tune your cache.

At runtime, use the **PolicyCenter Management Beans** page to enable the collection of detailed statistics for the global cache. If you disable the `GlobalCacheDetailedStats` parameter, PolicyCenter does not display the **Evict Information** and **Type of Cache Misses** graphs.

Default: False

GlobalCacheReapingTimeMinutes

Time (in minutes) since the last use before PolicyCenter considers cached items to be eligible for reaping. You can think of this as the period during which PolicyCenter is most likely to reuse an entity. See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 15

Minimum: 1

Maximum: 15

Set for Environment: Yes

GlobalCacheSizeMegabytes

The amount of space to allocate to the global cache. If you specify this value, it takes precedence over `GlobalCacheSizePercent`. See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Null: Yes

Set for Environment: Yes

GlobalCacheSizePercent

The percentage of the heap to allocate to the global cache. See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 15

Set for Environment: Yes

GlobalCacheStaleTimeMinutes

Time (in minutes) since the last write before PolicyCenter considers cached items to be stale and thus eligible for reaping. See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 60

Minimum: 1

Maximum: 120

Set for Environment: Yes

Can Change on Running Server: Yes

GlobalCacheStatsRetentionPeriodDays

Time period (in days), in addition to today, for how long PolicyCenter preserves statistics for historical comparison. See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 7

Minimum: 2

Maximum: 365

Set for Environment: Yes

GlobalCacheStatsWindowMinutes

Time period (in minutes). PolicyCenter uses this parameter for the following purposes:

- To define the period of time to preserve the reason for which PolicyCenter evicts an item from the cache, after the event occurred. If a cache miss occurs, PolicyCenter looks up the reason and reports the reason in the **Cache Info** page.
- To define the period of time to display statistics on the chart on the **Cache Info** page.

See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 30

Minimum: 2

Maximum: 120

Set for Environment: Yes

GroupCacheRefreshIntervalSecs

The time in seconds between refreshes of the group cache. This integer value must be zero (0) or greater. See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 600

ScriptParametersRefreshIntervalSecs

The time between refreshes of the script parameter cache, in seconds. This integer value must be zero (0) or greater. See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 600

TreeViewRefresh

The time in seconds that Guidewire PolicyCenter caches a tree view state.

Default: 120

ZoneCacheRefreshIntervalSecs

The time between refreshes of the zone cache, in seconds. See “Application Server Caching” on page 65 in the *System Administration Guide* for more information.

Default: 86400

Clustering Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to the application clusters.

To improve performance and reliability, you can install multiple PolicyCenter servers in a configuration known as a cluster. A cluster distributes client connections among multiple PolicyCenter servers, reducing the load on any one server. If one server fails, the other servers transparently handle its traffic.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

See also

- “Clustering Application Servers” on page 75 in the *System Administration Guide*

ClusteringEnabled

Whether to enable clustering on this application server.

Studio Read-only Mode

If you set the value of `ClusteringEnabled` to `true` in file `config.xml` on a particular application server and then restart the associated Studio, Studio becomes effectively read-only. In this context, *read-only* means:

- It is not possible to modify a Studio-managed resource. This applies, for example, to files that you open in the Gosu or Rules editor.
- If it is possible to modify a Studio-managed resource, it is not possible to save any modification you make to that resource. This applies, for example, to files that you open in the PCF editor.

To indicate the read-only status:

- Studio displays a padlock icon on the status bar that is visible only if Studio is in read-only mode. If you click the icon, Studio displays a modal message box indicating the reason why it is in read-only mode.
- Studio disables the **Save** button any time that Studio is in read-only mode.
- Studio changes the **Save** button tooltip in read-only mode to display the reason that save is not active in this mode. This is the same message that Studio shows if you click the padlock icon on the status bar.

Setting the value of configuration parameter `ResourcesMutable` to `false` provides the same Studio read-only behavior.

Default: False

Set for Environment: Yes

ClusterMemberPurgeDaysOld

The number of days to keep cluster member records before they can be deleted.

Default: 30

ClusterMemberRecordUpdateIntervalSecs

Cluster member database record update interval (in seconds). The same interval is used to reload the list of cluster members from the database. Note that this parameter is not used when JGroups-based implementation is used.

Default: 60

ClusterMulticastAddress

The IP multicast address to use in communicating with the other members of the cluster. This value must be unique within the range of the cache time-to-live parameter. This configuration parameter is meaningful only if configuration parameter `ClusteringEnabled` is set to `true`.

To be valid, a multicast address must be within the following specific range:

224.0.0.0 – 239.255.255.255

By convention, the Internet Assigned Numbers Authority (IANA) reserves certain addresses within the 224.0.x.x address space. See *IP4 Multicast Address Space Registry* at the following location for details:

<http://www.iana.org/assignments/multicast-addresses/multicast-addresses.xml>

Default: 228.9.9.9

Set for Environment: Yes

ClusterMulticastPort

The port used to communicate with other members of the cluster. This configuration parameter is meaningful only if configuration parameter `ClusteringEnabled` is set to `true`.

Default: 45566

Set for Environment: Yes

ClusterMulticastTTL

The time-to-live for cluster multicast packets. For Guidewire applications, this value is almost always 1, which means only deliver the packets to the local subnet. Higher time-to-live values require that you enable multicast routing on any intermediate routers (rare in most IT organizations). Also the larger the time-to-live value, the more you have to worry about allocating unique multicast addresses. This integer value must be zero (0) or greater. This configuration parameter is meaningful only if configuration parameter `ClusteringEnabled` is set to `true`.

Default: 1

Set for Environment: Yes

ClusterProtocolStackOption1

This is a local option that can contain other parameters for the `ClusterProtocolStack` stack string. You reference this option in the stack as `${option1}`. You configure this value in the default protocol stack in the UDP protocol. You can set it to the following so that a multi-home server can specify which NIC (Network Interface Card) to use for JGroups.

```
;bind_addr=xyz
```

Note: This string is a literal substitution. This requires that it start with the semicolon (;) UDP parameter delimiter. See the *JGroups* documentation for other values that you can assign to it.

To set the `bind_addr` bind address property for JBoss, you must start JBoss with the system property `-Dignore.bind.address=true`. See “Specifying the Bind Address” on page 21 in the *Installation Guide*.

Default: *None*

Set for Environment: Yes

ClusterProtocolStackOption2

This is a local option that can contain other parameters for the `ClusterProtocolStack` stack string. You reference this option in the stack as `${option2}`. See `ClusterProtocolStackOption1`.

Default: *None*

Set for Environment: Yes

ClusterProtocolStack

The cluster protocol stack string.

Default:

```
UDP{mcast_addr=${multicastAddress};ip_ttl=${timeToLive};mcast_port=${port}${option1}):
gw.JDBC_PING(timeout=5000;num_initial_members=4;num_ping_requests=3;updateInterval=30000):
MERGE3(max_interval=30000;min_interval=10000):
FD_ALL(interval=5000;timeout=27000):
VERIFY_SUSPECT(timeout=3000;num_msgs=3):
pbcast.NAKACK(retransmit_timeout=600,1200,2400,4800;discard_delivered_msgs=true):
UNICAST():
pbcast.STABLE(desired_avg_gossip=5000;max_bytes=4M):
FRAG:
pbcast.GMS(join_timeout=6000;merge_timeout=10000;print_local_addr=true)
```

ClusterStatisticsMonitorIntervalMins

Number of minutes between each cluster statistics monitor logging. A value of 0 means disable statistics logging. Note that this parameter is not used when JGroups-based implementation is used.

Default: 60

ConfigVerificationEnabled

Whether to check the configuration of this server against the other servers in the cluster. The default is `true`. You must also set configuration parameter `ClusteringEnabled` to `true` for `ConfigVerificationEnabled` to be meaningful. Do not disable this parameter in a production environment. Do not set this value to `false`, unless Guidewire Support specifically instructs you to do otherwise.

WARNING Guidewire specifically does not support disabling this parameter in a production environment. Do not set this parameter to `false` unless specifically instructed to do so by Guidewire Support.

Default: `True`

Set for Environment: Yes

JGroupsWatchdogHeartbeatIntervalSecs

The number of seconds between each heartbeat ping from the cluster coordinator to each of the nodes.

Default: 30

Set for Environment: No

JGroupsWatchdogMissedHeartbeatsBeforeReset

The number of missed heartbeats after which a node resets its JGroups channel.

Default: 10

Set for Environment: No

PDFMergeHandlerLicenseKey

Provides the BFO (Big Faceless Organization) license key needed for server-side PDF generation. If you do not provide a license key for a server, each generated PDF from that server contains a large DEMO watermark on its face. The lack of a license key does not, however, prevent document creation entirely.

It is possible to set this value differently for each server node in the cluster.

Default: None

Set for Environment: Yes

Database Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to the application database.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

DisableHashJoinPolicySearch

On an Oracle database, you can improve the performance of a policy search—if the search criteria include a first name, last name or a company name—by disabling a Hash Join.

Default: True

DisableIndexFastFullScanForPolicySearch

On Oracle, you can improve the performance of a policy search—if the search criteria include a first name, last name or a company name—by disabling an Index Fast Full Scan. This parameter has no effect on databases other than Oracle.

Default: True

DisableSortMergeJoinPolicySearch

On Oracle, you can improve the performance of a policy search—if the search criteria include a first name, last name or a company name—by disabling a Sort Merge Join. This parameter has no effect on databases other than Oracle.

Default: True

DiscardQueryPlansDuringStatsUpdateBatch

Whether to instruct the database to discard existing query plans during a database statistics batch process.

Default: False

IdentifyQueryBuilderViaComments

(SQL Server) Whether to provide comments with contextual information in certain SQL Select statements sent to the relational database. The comments are generated by instrumentation in higher level database objects created by using the query builder APIs.

The SQL comments are in the format:

```
/* applicationName:ProfilerEntryPoint */
```

The *applicationName* component of the comment is **PolicyCenter**.

The *ProfilerEntryPoint* component of the comment is the name of an entry point known to the Guidewire profiler for that application. For example, *ProfilerEntryPoint* might have the value `WebReq:ClaimSearch`.

Default: True

See also

- “Enabling Context Comments in Queries on SQL Server” on page 175 in the *Gosu Reference Guide*

IdentifyORMLayerViaComments

(SQL Server) Whether to provide comments with contextual information in certain SQL Select statements sent to the relational database. The comments are generated by instrumentation in lower level objects, such as beans, typelists, and other database building blocks.

The SQL comments are in the format:

```
/* applicationName:ProfilerEntryPoint */
```

The *applicationName* component of the comment is **PolicyCenter**.

The *ProfilerEntryPoint* component of the comment is the name of an entry point known to the Guidewire profiler for that application. For example, *ProfilerEntryPoint* might have the value `WebReq:ClaimSearch`.

Default: False

See also

- “Enabling Context Comments in Queries on SQL Server” on page 175 in the *Gosu Reference Guide*

MigrateToLargeIDsAndDatetime2

(SQL Server) Use to control whether to migrate to large (64-bit) IDs while upgrading the database. Migrating to large IDs is an expensive operation.

Default: False

Desktop and Team Parameters

Guidewire provides the following configuration parameters in `config.xml` that relate to the Desktop and Team tabs.

OtherWorkOrdersStatisticsWindowSize

Time window that the **Team** tab uses to calculate statistics for work orders other than renewals and submissions. Possible values are:

0	Use this week as the window, defined as the start of the current business week until now.
-1	Use this month as the window, defined as the start of the current month until now.
Any positive integer N	The window is the last N days of activity, including the current day.

Default: 0

See also

- “Setting the Window Size for Team Statistics” on page 516
- “Team Management” on page 673 in the *Application Guide*

RenewalsStatisticsWindowSize

Time window that the **Team** tab uses to calculate renewal statistics. Possible values are:

0	Use this week as the window, defined as the start of the current business week until now.
-1	Use this month as the window, defined as the start of the current month until now.
Any positive integer N	The window is the last N days of activity, including the current day.

Default: 0

See also

- “Setting the Window Size for Team Statistics” on page 516
- “Team Management” on page 673 in the *Application Guide*

SearchActivityThresholdDays

This parameter controls the threshold within which an activity must have been modified to qualify a job as being assigned to a user. The `UpdateTime` field on an activity contains a timestamp of when the activity was last modified. The activity must have been modified within `SearchActivityThresholdDays` days before the current date.

Default: 366

See also

- “Desktop Tab” on page 41 in the *Application Guide*
- “Team Tab” on page 48 in the *Application Guide*
- “Configuring the Team Tab” on page 515
- “Team Tab User Categories” on page 674 in the *Application Guide*

SubmissionsStatisticsWindowSize

Time window that the **Team** tab uses to calculate submission statistics. Possible values are:

0	Use this week as the window, defined as the start of the current business week until now.
-1	Use this month as the window, defined as the start of the current month until now.
Any positive integer N	The window is the last N days of activity, including the current day.

Default: 0

See also

- “Setting the Window Size for Team Statistics” on page 516
- “Team Management” on page 673 in the *Application Guide*

TeamScreenTabVisibilityRowCountCutoff

This parameter sets the maximum number of rows on the Team tab screens. Increasing this parameter potentially increases the amount of time that PolicyCenter takes to render the Team screens. If the results exceed the value of this parameter, then PolicyCenter displays a message on that screen and does not display the results.

If you select an individual user, PolicyCenter always displays the filtered results even if the number of results exceeds the cutoff parameter.

Default: 200

- “Team Tab” on page 48 in the *Application Guide*
- “Setting the Maximum Number of Rows on the Team Screens” on page 518

Document Creation and Document Management Parameters

Guidewire provides the following configuration parameters in the config.xml file that relate to document creation and management.

See also

- “Configuring Guidewire Document Assistant” on page 125 in the *System Administration Guide*.
- For information on editing config.xml and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

AllowDocumentAssistant

Whether to allow document management controls in the PolicyCenter interface. Setting this to false removes all controls from the interface, which results in reduced functionality. If false, this turns the Guidewire Document Assistant control off entirely and also forces the following parameters to be false:

- DisplayDocumentEditUploadButtons
- UseDocumentAssistantToDisplayDocuments

Default: false

DisplayDocumentEditUploadButtons

Whether the Documents list displays Edit and Upload buttons. Set this to false if the IDocumentContentSource integration mechanism does not support it.

Default: true

DocumentAssistantJNLP

The relative or absolute URL for the Document Assistant JNLP launch file.

Default: /jnlp/gw/documentassistant/DocumentAssistant.jnlp

DocumentContentDispositionMode

The Content-Disposition header setting to use any time that PolicyCenter returns document content directly to the browser. This parameter must be either `inline` or `attachment`.

Default: `inline`

DocumentTemplateDescriptorXSDLocation

The path to the XSD file that PolicyCenter uses to validate document template descriptor XML files. Specify this location relative to the following directory:

`modules/configuration/config/resources/doctemplates`

MaximumFileSize

The maximum allowable file size (in megabytes) that you can upload to the server. Any attempt to upload a file larger than this results in failure. Since the uploaded document must be handled on the server, this parameter protects the server from possible memory consumption problems.

Note: This parameter setting affects any imports managed through the PolicyCenter Administration tab. This specifically includes the import of administrative data and roles.

Default: 20

UseDocumentAssistantToDisplayDocuments

Whether to use the Guidewire Document Assistant control to display document contents.

Default: `true`

Domain Graph Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to the domain graph.

For information on editing and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

DomainGraphKnownLinksWithIssues

Use to define a comma-separated list of foreign keys. Each foreign key points from an entity outside of the domain graph to an entity inside the domain graph. Naming the foreign key in this configuration parameter suppresses the warning that would otherwise be generated for the link by the domain graph validator. Specify each foreign key on the list as the following:

`relative_entity_name:foreign_key_property_name`

IMPORTANT You are responsible for assuring these foreign keys are `null` at the time PolicyCenter is ready to archive the graph.

Default: None

DomainGraphKnownUnreachableTables

Use to define a comma-separated list of relative names of entity types that are linked to the graph through a nullable foreign key. This can be problematic because the entity can become unreachable from the graph if the foreign key is `null`. Naming the type in this configuration parameter suppresses the warning that would otherwise be generated for the type by the domain graph validator

Default: None

Environment Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to the application environment.

For information on editing and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

AddressVerificationFailureAsError

Set to `true` to have address verification failures shown as errors instead of warnings. This parameter is meaningless if `EnableAddressVerification` is set to `false`.

This parameter works in concert with `EnableAddressVerification`. For more information, see “`EnableAddressVerification`” on page 53.

Default: `false`

AlwaysShowPhoneWidgetRegionCode

Whether the phone number widget in the application user interface always displays a selector for phone region codes.

Default: `false`

Set for Environment: Yes

CurrentEncryptionPlugin

Set this value to the name of the plugin that you intend to use to manage encryption. Typically, a Guidewire installation has only a single implementation of an encryption plugin interface. However, you can, for example, decide to implement a different encryption algorithm using a different implementation of the encryption interface as part of an upgrade process. In this case, you must retain your old encryption plugin implementation in order to support the upgrade.

To support multiple implementations of encryption plugins, PolicyCenter provides the `CurrentEncryptionPlugin` configuration parameter. Set this configuration parameter to the `EncryptionID` of the encryption plugin currently in use—if you have implemented multiple versions `IEncryption` plugin interface.

- If you do not provide a value for this configuration parameter, then data is unencrypted.
- If you create multiple implementations of a plugin interface, then you must name each plugin implementation individually and uniquely.

IMPORTANT PolicyCenter does not provide an encryption algorithm. You must determine the best method to encrypt your data and implement it.

Default: None

See also

- For information on the how to configure your database to support encryption, see “Encryption Integration Overview” on page 239 in the *Integration Guide*.
- For information on the steps to take if you upgrade your installation and change your encryption algorithm, see “Changing Your Encryption Algorithm Later” on page 244 in the *Integration Guide*.
- For information on using the Plugins Registry editor, see “Using the Plugins Registry Editor” on page 109.

DeprecatedEventGeneration

Whether to use the now-deprecated event logic that had previously been available.

Default: False

EnableAddressVerification

Set this value to `true` to enable address verification warnings. Address verification checks that all the fields match each other based on the zone data.

This parameter works in concert with the `AddressVerificationFailureAsError` parameter to show either a warning or an error, as follows:

- If `EnableAddressVerification` is `true` and `AddressVerificationFailureAsError` is `false`, ClaimCenter shows a warning message if verification against zone data fails.
- If `EnableAddressVerification` is `true` and `AddressVerificationFailureAsError` is `true`, ClaimCenter shows an error message if verification against zone data fails.
- If `EnableAddressVerification` is `false`, ClaimCenter does not verify the address based on zone data.

Default: False

See also

- “`AddressVerificationFailureAsError`” on page 52

EnableInternalDebugTools

Make internal debug tools available to developer.

Default: False

Set for Environment: Yes

KeyGeneratorRangeSize

The number of key identifiers (as a block) that the server obtains from the database with each request. This integer value must be zero (0) or greater.

Default: 100

MemoryUsageMonitorIntervalMins

How often the PolicyCenter server logs memory usage information, in minutes. This is often useful for identifying memory problems.

To disable the memory monitor, do one of the following:

- Set this parameter to 0.
- Remove this parameter from `config.xml`.

Default: 0

PublicIDPrefix

The prefix to use for public IDs generated by the application. Generated public IDs are of the form *prefix: id*. This *id* is the actual entity ID. Guidewire strongly recommends that you set this parameter to different values in production and test environments to allow for the clean import and export of data between applications.

This `PublicIDPrefix` must not exceed 9 characters in length. Use only letters and numbers. Do not use space characters, colon characters, or any other characters that other applications might process or escape specially. Do not specify a two-character value. Guidewire reserves for itself all public IDs that start with a two-character ID and then a colon

IMPORTANT Guidewire reserves two-character public ID prefixes for its own current or future use.

Required: Yes

Default: *None*

ResourcesMutable

Indicates whether resources are mutable (modifiable) on this server. If you connect Studio to a remote server (on which this parameter is set to `true`), then Studio pushes resource changes to the remote server as you save local changes. Guidewire strongly recommends that you set this value to `false` on a production server to prevent changes to the configuration resources directory.

See also “[RetainDebugInfo](#)” on page 55.

Studio Read-only Mode

If you set the value of `ResourcesMutable` to `false` in `config.xml` on a particular application server and then restart the associated Studio, that Studio becomes effectively read-only. In this context, read-only means:

- It is not possible to modify a Studio-managed resource. This applies, for example, to files that you open in the Gosu or Rules editor.
- If it is possible to modify a Studio-managed resource, it is not possible to save any modification you make to that resource. This applies, for example, to files that you open in the PCF editor.

To indicate the read-only status:

- Studio displays a padlock icon on the status bar that is visible only if Studio is in read-only mode. If you click the icon, Studio displays a modal message box indicating the reason why it is in read-only mode.
- Studio disables the **Save** button any time that Studio is in read-only mode.
- Studio changes the **Save** button tooltip in read-only mode to display the reason that save is not active in this mode. This is the same message that Studio shows if you click the padlock icon on the status bar.

Setting the value of configuration parameter `ClusteringEnabled` to `true` provides the same Studio read-only behavior.

Default: True

WARNING Guidewire recommends that you always set this configuration parameter to `false` in a production environment. Setting this parameter to `true` has the potential to modify resources on a production server in unexpected and possibly damaging ways.

RetainDebugInfo

Whether the production server retains debugging information. This parameter facilitates debugging from Studio without a type system refresh.

- If set to `true`, PolicyCenter does not clear debug information after compilation.
- If set to `false`, the server does not retain sufficient debugging information to enable debugging. As a production server does not recompile types, it is not possible to regenerate any debugging information.

Default: `False`

See also

“[ResourcesMutable](#)” on page 54.

StrictDataTypes

Controls whether PolicyCenter uses the pre-PolicyCenter 4.0 behavior for configuring data types, through the use of the `fieldvalidators.xml` file.

- Set this value to `false` to preserve the existing behavior. This is useful for existing installations that are upgrading but want to preserve the existing functionality.
- Set this value to `true` to implement the new behavior. This is use for new PolicyCenter installations that want to implement the new behavior.

StrictDataTypes = true

If you set the `StrictDataTypes` value to `true`, then PolicyCenter:

- Does not permit decimal values to exceed the scale required by the data type. The setter throws a `gw.datatype.DataTypeException` if the scale is greater than that allowed by the data type. You are responsible for rounding the value, if necessary.
- Validates field validator formats in both the PolicyCenter user interface and in the field setter.
- Validates numeric range constraints in both the PolicyCenter user interface and in the field setter.

StrictDataTypes = false

If you set the `StrictDataTypes` value to `false`, then PolicyCenter:

- Auto-rounds decimal values to the appropriate scale, using the `RoundHalfUp` method. For example, setting the value 5.045 on a field with a scale of 2 sets the value to 5.05.
- Validates field validator formats in the interface but not at the setter level. For example, PolicyCenter does not permit a field with a validator format of `[0-9]{3}-[0-9]{2}-[0-9]{4}` to have the value 123-45-A123 in the interface. It is possible, however, to set a field to that value in Gosu code, for example. This enables you to bypass the validation set in the interface.
- Validates numeric range constraints in the interface, but not at the setter level. For example, Guidewire does not allow a field with a maximum value of 100 to have the value 200 in the interface. However, you can set the field to this value in Gosu rules, for example. This enables you to bypass the validation set in the interface.

Default: `True`

TwoDigitYearThreshold

The threshold year value for determining whether to resolve a two-digit year to the earlier or later century.

Default: `50`

UnreachableCodeDetection

Determines whether the Gosu compiler generates errors if it detects unreachable code or missing return statements.

Default: True

UnrestrictedUserName

The `username` of the user who has unrestricted permissions to do anything in PolicyCenter.

Default: su

UseOldStylePreUpdate

Setting this parameter to `true` is deprecated in PolicyCenter. The base configuration contains no Preupdate rules, therefore setting this parameter to true has no effect.

Default: False

WarnOnImplicitCoercion

A value of `true` indicates that the Gosu compiler generates warnings if it determines that an implicit coercion is occurring in a program.

Default: True

WebResourcesDir

Specifies the location of the Web resources directory under the root of the Tomcat configuration directory.

Default: resources

Financial Parameters

Guidewire provides the following parameters in the `config.xml` file to help configure how PolicyCenter works with monetary amounts.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

See also

- “Configuring Currencies” on page 109 in the *Globalization Guide*

Geocoding Feature Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to geocoding.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

UseGeocodingInPrimaryApp

If `true`, PolicyCenter enables searching for nearby locations in the Reinsurance Management user interface. ContactManager does not respond to this parameter.

Default: `False`

ProximitySearchOrdinalMaxDistance

The maximum distance to use if performing an *ordinal* (nearest *n* items) proximity search. This distance is in miles, unless you set `UseMetricDistancesByDefault` to `true`. This parameter has no effect on *radius* (within *n* miles or kilometers) proximity searches or walking-the-group-tree-based proximity assignment.

Default: `300`

ProximityRadiusSearchDefaultMaxResultCount

The maximum number of results to return if performing a *radius* (within *n* miles or kilometers) proximity search. This parameter has no effect on *ordinal* (nearest *n* items) proximity searches.

Default: `1000`

UseMetricDistancesByDefault

If `true`, PolicyCenter uses kilometers and metric distances instead of miles and United States distances for location searches. Set this parameter identically in both PolicyCenter and ContactManager.

Default: `False`

Globalization Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to globalization.

The globalization parameters include:

- `DefaultApplicationLanguage`
- `DefaultApplicationLocale`
- `DefaultApplicationCurrency`
- `DefaultRoundingMode`
- `MulticurrencyDisplayMode`
- `DefaultCountryCode`
- `DefaultPhoneCountryCode`
- `DefaultNANPACountryCode`
- `AlwaysShowPhoneWidgetRegionCode`

IMPORTANT If you integrate the core applications in Guidewire InsuranceSuite, you must set the values of `DefaultApplicationCurrency` and `MulticurrencyDisplayMode` to be the same in each application.

See also

- For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

DefaultApplicationLanguage

Default display language for the application as a whole.

IMPORTANT This parameter setting is permanent. Once you set the parameter and then start the server, you cannot change the value.

Default: en_US

Set for Environment: Yes

Permanent: Yes.

See also

- “Setting the Default Display Language” on page 33 in the *Globalization Guide*

DefaultApplicationLocale

Default locale for regional formats in the application. You must set configuration parameter `DefaultApplicationLocale` to a typecode contained in the `LocalType` typelist.

IMPORTANT This parameter setting is permanent. Once you set the parameter and then start the server, you cannot change the value.

Default: en_US

Set for Environment: Yes

Permanent: Yes

See also

- “Setting the Default Application Locale for Regional Formats” on page 84 in the *Globalization Guide*

DefaultApplicationCurrency

Default currency for the application. You must set configuration parameter `DefaultApplicationCurrency` to a typecode contained in the `Currency` typelist, even if you configure PolicyCenter with a single currency. Guidewire applications which share currency values must have the same `DefaultApplicationCurrency` setting in their respective `config.xml` files. The default currency is sometimes known as the *preferred* currency.

IMPORTANT This parameter setting is permanent. Once you set the parameter and then start the server, you cannot change the value.

Default: usd

Set for Environment: Yes

Permanent: Yes

See also

- “Setting the Default Application Currency” on page 114 in the *Globalization Guide*

DefaultRoundingMode

Sets the default rounding mode for monetary amount calculations. The available choices are a subset of those supported by `java.math.RoundingMode`, namely:

- UP
- DOWN
- CEILING

- FLOOR
- HALF_UP
- HALF_DOWN
- HALF_EVEN

Guidewire strongly recommends that you use one of the following:

- HALF_UP
- HALF_EVEN

You can access this value in Gosu code by using the following method:

```
gw.api.util.CurrencyUtil.getRoundingMode()
```

IMPORTANT This parameter setting is permanent. Once you set the parameter and then start the server, you cannot change the value.

Default: HALF_UP

Permanent: Yes

See also

- “Choosing a Rounding Mode” on page 115 in the *Globalization Guide*

MulticurrencyDisplayMode

Determines whether PolicyCenter displays currency selectors for monetary values. You can set

`MulticurrencyDisplayMode` to one of the following values:

- SINGLE
- MULTIPLE

In the base configuration of PolicyCenter, the value is set to SINGLE. If you want your configuration to support multiple currencies, you must change the value `MulticurrencyDisplayMode` before you start the PolicyCenter server for the first time. If you change the value to MULTIPLE after the server starts for the first time, subsequent attempts to start the server fail.

Default: SINGLE

Permanent: Yes

See also

- See “Setting the Currency Display Mode” on page 115 in the *Globalization Guide*.

DefaultCountryCode

The default ISO country code to use if the country for address is not set explicitly. PolicyCenter uses this also as the default for new addresses that it creates.

See the following for a list of valid ISO country codes:

http://www.iso.org/iso/english_country_names_and_code_elements

DefaultPhoneCountryCode

The default ISO country code used for phone data.

Default: None

DefaultNANPACountryCode

The default country code for region 1 phone numbers. If the area code is not in the `nanpa.properties` map file, then it defaults to the value configured with this parameter.

Default: US

AlwaysShowPhoneWidgetRegionCode

Whether the phone number widget in the application user interface always displays a selector for phone region codes.

Default: false

Integration Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to how multiple Guidewire applications integrate with each other.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

BillingSystemURL

URL to use in `ExitPoint` PCF pages that view items in the billing system.

- If integrating Guidewire PolicyCenter with Guidewire BillingCenter, then set this parameter to the BillingCenter base URL (for example, `http://server/bc`). In this case, the exit points add the appropriate BillingCenter entry point.
- If integrating with a non-Guidewire billing system, then you need to modify the `ExitPoint` PCF to set up the parameters required by that system.
- If you omit this parameter or if you set it to an empty string, then PolicyCenter hides the buttons in the interface that take you to the exit points.

Default: Empty string

See also

- “PolicyCenter Exit Points to ClaimCenter and BillingCenter” on page 510 in the *Integration Guide*

ClaimSystemURL

URL to use in `ExitPoint` PCF pages that view items in the claims system.

- If integrating Guidewire PolicyCenter with Guidewire ClaimCenter, then set this parameter to the ClaimCenter base URL (for example, `http://server/cc`). In this case, the exit points add the appropriate ClaimCenter entry point.
- If integrating with a non-Guidewire claim system, then you need to modify the `ExitPoint` PCF to set up the parameters required by that system.
- If you omit this parameter or if you set it to an empty string, then PolicyCenter hides the buttons in the interface that take you to the exit points.

Default: Empty string

See also

- “PolicyCenter Exit Points to ClaimCenter and BillingCenter” on page 510 in the *Integration Guide*

DefaultXmlExportEncryptionId

The unique encryption ID of an encryption plugin. If archiving is enabled, PolicyCenter uses that encryption plugin to encrypt any encrypted fields during XML serialization.

Default: null (no encryption)

KeepCompletedMessagesForDays

Number of days after which PolicyCenter purges old messages in the message history table.

Default: 90

LockPrimaryEntityDuringMessageHandling

If it is set to true, PolicyCenter locks the primary entity associated with a message at the database level during the following operations:

- During a message send operation
- During message reply handling
- During marking a message as skipped

If the message has no primary entity associated with it, then this configuration parameter has no effect.

Default: true

PaymentSystemURL

URL to use in `ExitPoint` PCF pages that view items in a payment system.

- If integrating Guidewire PolicyCenter with Guidewire BillingCenter, then set this parameter to the BillingCenter base URL (for example, `http://server/bc`). In this case, the exit points add the appropriate BillingCenter entry point.
- If integrating with a non-Guidewire billing system, then you need to modify the `ExitPoint` PCF to set up the parameters required by that system.
- If you omit this parameter or if you set it to an empty string, then PolicyCenter hides the buttons in the interface that take you to the exit points.

Note: Guidewire configures this parameter in the base configuration to work with a demonstration payment system. For more details, see “Implementing the Billing Summary Plugin” on page 497 in the *Integration Guide*.

Default: pc

See also

- “PolicyCenter Exit Points to ClaimCenter and BillingCenter” on page 510 in the *Integration Guide*

PluginStartupTimeout

OSGi plugins startup timeout (in seconds). The `PluginConfig` component waits for at most this time for all required OSGi plugins to start. The `PluginConfig` component reports an error for each OSGi plugin that does not start after this timeout has expired.

Default: 60

Job Expiration Parameters

Guidewire provides the following configuration parameters in `config.xml` that relate how PolicyCenter handles jobs.

PolicyCenter provides a **Job Expire** (`jobexpire`) batch process that moves jobs from the **New**, **Draft**, or **Quote** status to the **Expired** status. This process first examines all jobs that have a **New**, **Draft**, or **Quote** status. From this set of jobs, the batch process expires the jobs that meet all of the following criteria:

- The job meets the expiration threshold.
- The job meets the creation threshold.
- The `job.canExpireJob` method returns `true` for that job type.

In the base configuration, this process moves Submission jobs to the **Expired** status if all of the following are true:

- The job status is either **New**, **Draft**, or **Quote**.
- The job is at least seven days past the effective date of the policy.

Setting the expiration and creation thresholds – Modify the following parameters to configure the expiration and creation thresholds. If you enable both parameters, the job must meet both thresholds.

- `JobExpirationEffDateThreshold` – The number of days past the policy effective date before the batch process `JobExpire` expires a job.
- `JobExpirationCreateDateThreshold` – The number of days past the job creation date before batch process `JobExpire` expires a job.

Expiring different job types – You can also enable expiration for job types other than Submission. To facilitate this process, PolicyCenter provides a number of Boolean configuration parameters of the following pattern:

`JobExpirationCheck<JobType>`

Job types – For this configuration parameter, `<JobType>` can be any of the following:

- Audit
- Cancellation
- Issuance
- PolicyChange
- Reinstatement
- Renewal
- Rewrite
- Submission
- TestJob

To enable the expiration of a particular job type, set `JobExpirationCheck<JobType>` to `true` for that particular job type.

See also

- “[Changing Jobs to Expired Status](#)” on page 592
- “[Scheduling Work Queues and Batch Processes](#)” on page 107 in the *System Administration Guide*

[`JobExpirationCheckAudit`](#)

Set to `true` to enable PolicyCenter to expire this job type. For audit jobs, you must override the `canExpireJob` method in the `gw.job.AuditProcces` class and modify it to return `true`. You can find this class in the `Classes` folder in Studio.

Default: `False`

JobExpirationCheckCancellation

Set to `true` to enable PolicyCenter to expire this job type.

Default: `False`

JobExpirationCreateDateThreshold

Number of days past the job creation date before batch process `JobExpire` expires a job. Batch process `JobExpire` first examines all jobs meeting the date criteria set by this parameter. In addition to this threshold, the job must also meet the `JobExpirationEffDateThreshold` threshold. The batch process then expires those jobs for which `job.canExpireJob` is `true`.

The job creation date is specified in `Job.CreateTime`.

A value of `-1` disables this parameter.

IMPORTANT Setting the create date threshold to a negative number effectively disables create date checking, as it is not possible to create a job with a future create date. Disabling `JobExpirationCreateDateThreshold` does not disable `JobExpirationEffDateThreshold`.

Default: `-1`

JobExpirationEffDateThreshold

Number of days past the job effective date before batch process `JobExpire` expires a job. Batch process `JobExpire` first examines all jobs meeting the date criteria set by this parameter. In addition to this threshold, the job must also meet the `JobExpirationCreateDateThreshold` threshold. The batch process then expires those jobs for which `job.canExpireJob` is `true`.

The job effective date is the `EditEffectiveDate` of a `PolicyPeriod` on the `Job`. If the job has more than one policy period, then each active policy period on the job must meet the threshold. You access the active policy periods on the job through the `Job.ActivePeriods` array.

A negative value means that the batch process expires unbound jobs by that number of days before their effective date is reached. For example, a value of `-5` for this parameter indicates that jobs expire five days before their effective date.

Default: `7`

JobExpireCheckIssuance

Set to `true` to enable PolicyCenter to expire this job type.

Default: `False`

JobExpireCheckPolicyChange

Set to `true` to enable PolicyCenter to expire this job type.

Default: `False`

JobExpireCheckReinstatement

Set to `true` to enable PolicyCenter to expire this job type.

Default: `False`

JobExpireCheckRenewal

Set to `true` to enable PolicyCenter to expire this job type.

Default: False

JobExpireCheckRewrite

Set to `true` to enable PolicyCenter to expire this job type.

Default: False

JobExpireCheckSubmission

Set to `true` to enable PolicyCenter to expire this job type.

Default: True

JobExpireCheckTestJob

Set to `true` to enable PolicyCenter to expire this job type.

Default: False

Lookup Table Parameters

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

AvailabilityContextCacheSize

The maximum size of the cache holding `AvailabilityContexts` after a lookup.

Default: 1000

Required: Yes

Miscellaneous Job-Related Parameters

Guidewire provides the following configuration parameters in `config.xml` that relate to PolicyCenter jobs.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

AllowedDaysBeforeOrAfterPolicyStartDate

Number of days, before or after a given policy period `StartDate`, to search for a non-cancelled bound `PolicyPeriod` on which to base the renewal.

For example, if you need to rewrite a cancelled policy with an earlier start date, the most recent bound period is the cancelled period. To renew such a policy, it must be possible to identify the rewrite. This parameter establishes the windows in which to search for a rewritten policy.

- If the policy term is 6 months or greater, then the default of 90 days is acceptable.
- If the policy term is shorter, then use a smaller number for this parameter.

Default: 90

BoundPolicyThresholdDays

Minimum number of days that must pass before the Exception rules run again on a bound PolicyPeriod. This integer value must be zero (0) or greater.

Default: 14

Change on Running Server: Yes

ClosedPolicyThresholdDays

Minimum number of days that must pass before the Exception rules run again on a closed PolicyPeriod. This integer value must be zero (0) or greater.

Default: 14

Change on Running Server: Yes

MaximumPolicyCreationYearDelta

Value, which added to the current year, represents the maximum year to use during policy creation. This integer value must be zero (0) or greater.

Default: 1

MaxRecentAccounts

Maximum number of recent accounts that PolicyCenter shows in the Account tab.

Default: 5

MaxRecentPoliciesAndJobs

Maximum number of recent policies and jobs that PolicyCenter shows in the Policy tab.

Default: 8

MaxSubmissionsToCreate

Maximum number of submissions that PolicyCenter can create at one time in the New Submission page.

Default: 20

MinimumPolicyCreationYear

Minimum (earliest) year to use during policy creation. This integer value must be 1000 or greater.

Default: 1000

OpenPolicyThresholdDays

Minimum number of days that must pass before the Exception rules run again on an open PolicyPeriod. This integer value must be zero (0) or greater.

Default: 1

Change on Running Server: Yes

PatternCacheMaxDuration

Upper bound on how long PolicyCenter allows caches of pattern entities to exist without refresh, measured in seconds.

Default: 86400

PolicyChangeMaxQuotes

Maximum number of allowable quotes for a policy change.

Default: 3

RenewalMaxQuotes

Maximum number of allowable quotes for a renewal.

Default: 3

RenewalProcessLeadTime

Lead time (in number of days) before the state non-renewal notification deadline to start renewals.

Default: 165

SubmissionMaxQuotes

Maximum number of allowable quotes for a submission.

Default: 3

Miscellaneous Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to various miscellaneous application features.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

ActivityStatisticsWindowSize

Time window to capture activity statistics.

Default: 0

ConsistencyCheckerThreads

Number of threads to use when running the consistency checker.

Default: 1

DefaultDiffDateFormat

Sets the default format for dates in the difference tree user interface. Valid formats mirror the Java default date formats:

- short
- medium

- long
- full

Default: short

DisableDomainGraphSupport

All PolicyCenter environments are expected to have valid data models even if they are not currently using a feature that requires domain graph support. Archiving and quote purging require domain graph support.

In rare situations, you may wish to upgrade a PolicyCenter environment that does not yet have a valid data model. If there are compelling business reasons not to correct the data model during the upgrade, you can disable domain graph support using the `DisableDomainGraphSupport` parameter. If you set this parameter to `true`, certain features, including archiving and quote purging, will fail ungracefully.

Guidewire strongly recommends that you resolve domain graph issues at the earliest opportunity to minimize the upgrade cost of the changes required to achieve a valid data model.

Default: false

IgnoreLeapDayForEffDatedCalc

In the default configuration, the `IgnoreLeapDayForEffDatedCalc` parameter determines whether PolicyCenter ignores a leap day in the following calculations:

- Calculating scalable properties
- Prorating premiums for policies that are less than a full term

The following table describes the values for the `IgnoreLeapDayForEffDatedCalc` parameter:

Value	Description
true	PolicyCenter does not include the leap day while calculating the values for scalable properties or prorated premiums.
false	PolicyCenter includes the leap day while calculating the scalable property amount or prorated premium.

In the default configuration, some scalable properties are `OverrideAmount` on the `Cost` entity and `ScalableBasisAmount` on the `GLExposure` entity.

Default: true

Leap Days and Prorated Premiums Example

Assume that PolicyCenter needs to rate a policy change that affects the entire month of February in a year that includes a leap day. If `IgnoreLeapDayforEffDatedCalc` is `true`, PolicyCenter ignores the leap day and prorates the premium for 28 days. If `false`, PolicyCenter includes the leap day and prorates the premium for 29 days.

PolicyCenter prorates the premium in the proration plugin. It is possible to override the `IgnoreLeapDayForEffDatedCalc` parameter by a parameter on the plugin. For more information, see “Proration Plugin” on page 169 in the *Integration Guide*.

InitialSampleDataSet

Optional set of sample data to load at server start up. This value must be a typecode from the `SampleDataSet` typelist, or `null`. In the base application configuration, valid typecodes are:

Tiny	Just a few users like <code>aapplegate</code> , good for test purposes.
Small	A small community model and a few sample Accounts and Policies, useful for local development.
Large	A large, full data set useful for application demonstrations and manual Quality Assurance.

Set value to `Tiny`, `Small`, or `Large`, corresponding to the data set that you want to load:

- If there is no sample data loaded as the server starts, PolicyCenter loads the sample data set that you specify.
- If there is sample data, PolicyCenter does nothing.

Default: `None`

Set for Environment: Yes

JGroupsClusterChannel

Whether to use JGroups based cluster channel.

Default: `true`

ListViewPageSizeDefault

The default number of entries that PolicyCenter displays in each page in a list view, if the page does not explicitly specify this value. This integer value must be at least 1.

Default: 15

Minimum: 1

ProfilerDataPurgeDaysOld

Number of days to keep profiler data before PolicyCenter deletes it.

Default: 30

TransactionIdPurgeDaysOld

Number of days to keep external transaction ID records before they can be deleted.

Default: 30

PDF Print Settings Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to the generation of PDF files from PolicyCenter.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

DefaultContentDispositionMode

The Content-Disposition header setting to use any time that PolicyCenter returns document content directly to the browser. PolicyCenter uses this setting for content, such as exports or printing, but not for documents. This parameter must be either `inline` or `attachment`.

Default: attachment

PrintFontFamilyName

Use to configure FOP settings for printing non-U.S. character sets. (FOP refers to the Apache Formatting Objects Processor.) Set this value to the name of the font family for custom fonts as defined in your FOP user configuration file. For more information, refer to the following:

<http://xmlgraphics.apache.org/fop/>

Default: san-serif

PrintFontSize

Font size of standard print text.

Default: 10pt

PrintFOPUserConfigFile

Path to FOP user configuration file, which contains settings for printing non-U.S. character sets. (FOP refers to the Apache Formatting Objects Processor.) Enter a fully qualified path to a valid FOP user configuration file. There is no default. However, a typical value for this parameter is the following:

C:\fopconfig\fop.xconf

For more information, refer to the following:

<http://xmlgraphics.apache.org/fop/>

Default: None

PrintHeaderFontSize

Font size of headers in print text.

Default: 16pt

PrintLineHeight

Total size of a line of print text.

Default: 14pt

PrintListViewBlockSize

Use to set the number of elements in a list view to print as a block. This parameter splits the list into blocks of this size, with a title page introducing each block of elements. A large block size consumes more memory during printing, which can cause performance issues. For example, attempting to print a block of several thousand elements can potentially cause an out-of-memory error.

Default: 20

PrintListViewFontSize

Font size of text within a list view.

Default: 10pt

PrintMarginBottom

Bottom margin size of print page.

Default: 0.5in

PrintMarginLeft

Left margin size of print page.

Default: 1in

PrintMarginRight

Right margin size of print page.

Default: 1in

PrintMarginTop

Top margin size of print page.

Default: 0.5in

PrintMaxPDFInputFileSize

During PDF printing, PolicyCenter first creates an intermediate XML file as input to a PDF generator. If the input is very large, the PDF generator can run out of memory.

Value	Purpose
Negative	A negative value indicates that there is no limit on the size of the XML input file to the PDF generator.
Non-negative	A non-negative value limits the size of the XML input file to the set value (in megabytes). If a user attempts to print a PDF file that is larger in size than this value, PolicyCenter generates an error.

Default: -1

PrintPageHeight

Total print height of page.

Default: 8.5in

PrintPageWidth

Total print width of page.

Default: 11in

Product Model

Guidewire provides the following configuration parameters in `config.xml` that relate to the PolicyCenter product model.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

ExternalProductModelDirectory

PolicyCenter loads the directory specified by `ExternalProductModelDirectory` parameter at both server startup time and when a reload is requested if:

- The parameter specifies a directory accessible to the application server.
- The directory is not in the deployment directory.
- The directory contains a complete product model definition that only defines existing patterns.
- The directory contains at least one lookup XML file ending in `-lookups.xml`.
- The files in the directory are valid.

Otherwise, PolicyCenter loads availability data from the standard configuration module at server startup time. An attempt to reload will fail.

Default: None

Can Change on Running Server: No

See also

- “Reloading Availability Data” on page 95 in the *Product Model Guide*

Quote Purging Configuration Parameters

This topic describes quote purging configuration parameters related to *purgings* jobs and *pruning* policy periods.

Purging removes jobs after a specified length of time has passed. The jobs must meet the purging criteria.

Pruning removes unselected policy periods attached to a job. The selected policy period is specified by the `SelectedVersion` property on the `Job` entity. Unselected policy periods are any policy periods attached to the job that are not the `SelectedVersion`.

See also

- “Quote Purging” on page 445 in the *Application Guide*
- “Configuring Quote Purging” on page 457

PruneAndPurgeJobsEnabled

Specifies whether the Purge batch process is enabled.

Default: false

See also

- “Enabling Quote Purging Batch Processes” on page 461

PruneVersionsDefaultRecheckDays

The minimum number of days that must pass after failing a purging check before reconsidering the unselected policy periods on the job for pruning.

Default: 30

PruneVersionsPolicyTermDays

The minimum number of days that must pass after a job's selected policy period has ended before the unselected policy periods on the job are eligible for pruning. This parameter is ignored if `PruneVersionsPolicyTermDaysCheckDisabled` is `true`.

Default: 30

PruneVersionsPolicyTermDaysCheckDisabled

If `true`, then disable the `PruneVersionsPolicyTermDays` parameter. Setting this parameter to `true` does not disable pruning closed jobs.

Default: `false`

See also

- “Get Prune Job Date” on page 177 in the *Integration Guide*

PruneVersionsRecentJobCompletionDays

The minimum number of days that must pass after a job is closed before the unselected policy periods on the job are eligible for pruning.

Default: 210

PurgeJobsDefaultRecheckDays

The minimum number of days that must pass after failing a purging check before a job is reconsidered for purging.

Default: 30

PurgeJobsPolicyTermDays

The minimum number of days that must pass after the end of a job's selected policy period before the job is eligible for purging. This parameter is ignored if `PurgeJobsPolicyTermDaysCheckDisabled` is `true`.

Default: 30

PurgeJobsPolicyTermDaysCheckDisabled

If `true`, then disable the `PurgeJobsPolicyTermDays` parameter. Setting this parameter to `true` does not disable purging closed jobs.

Default: `false`

See also

- “Get Purge Job Date” on page 177 in the *Integration Guide*

PurgeJobsRecentJobCompletionDays

The minimum number of days that must pass after a job is closed before the job is eligible for purging.

Default: 210

PurgeOrphanedPolicyPeriodsEnabled

Specifies whether the Purge Orphaned Policy Periods batch process is enabled.

Default: false

See also

- “Enabling Quote Purging Batch Processes” on page 461

Rating Management Parameters

Guidewire provides the following configuration parameters in the config.xml file that relate to 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.

PurgeWorksheetsEnabled

Enable the Purge Rating Worksheets batch process.

Default: false

See also

- “Extract Rating Worksheets Batch Process” on page 544

RateRoutineIndexingThreshold

When editing a long rate routine, you can edit the rate routine by section. A long rate routine has more steps than the value of the indexing threshold, theRateRoutineIndexingThreshold parameter.

Default: 150

See also

- “Editing Long Rate Routines” on page 590 in the *Application Guide*

RateTableManagementNormalizationRowLimit

PolicyCenter determines whether to normalize a rate table if the table contains this many rows or more.

If the number of rows in the normalized table would exceed this limit, the table is automatically marked as non-normalizable. PolicyCenter stores the non-normalized version of the table.

Default: 10000

See also

- “Rate Table Normalization Configuration Parameters” on page 525

RateTableManagementNormalizationRowThreshold

If the number of rows in the normalized table would exceed this threshold, the user is given the option to store a non-normalized version of the table.

Default: 1000

See also

- “Rate Table Normalization Configuration Parameters” on page 525

RatingWorksheetContainerAgeForPurging

The Purge Rating Worksheets batch process uses this parameter as one factor in determining if a worksheet container can be purged. To be considered for purging, the worksheet container’s job must have been completed at least RatingWorksheetContainerAgeForPurging days.

Default: 90

See also

- “Purge Rating Worksheet Batch Process” on page 544

Scheduler and Workflow Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to batch process scheduler and workflow.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

SchedulerEnabled

Whether to enable the internal batch process application scheduler. See “Batch Processing” on page 99 in the *System Administration Guide* for more information on batch processes and the scheduler.

Default: True

Can Change on Running Server: Yes

WorkflowLogDebug

Configuration parameter `WorkflowLogDebug` takes a Boolean value:

- If set to `true`, PolicyCenter outputs the ordinary verbose system workflow log messages from the Guidewire server to the workflow log.
- If set to `false`, PolicyCenter does not output any of the ordinary system messages.

The setting of this parameter does not have any effect on calls to log workflow messages made by customers. Therefore, all customer log messages are output. If customers experience too many workflow messages being written to the `xx_workflowlog` table, Guidewire recommends that you set this parameter to `false`.

Default: True

WorkflowLogPurgeDaysOld

Number of days to retain workflow log information before PolicyCenter deletes it.

Default: 30

WorkflowPurgeDaysOld

Number of days to retain workflow information before PolicyCenter deletes it.

Default: 60

WorkflowStatsIntervalMins

Aggregation interval in minutes for workflow timing statistics. Statistics such as the mean, standard deviation, and similar statistics used in reporting on the execution of workflow steps all use this time interval. A value of 0 (zero) disables statistics reporting.

Default: 60

Search Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to searching.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

ContactSearchMaxResult

Number of maximum results to return from a contact search.

Default: 100

FreeTextSearchEnabled

Whether to enable the free-text search feature. Setting the `FreeTextSearchEnabled` parameter to `true` enables the free-text plugins for indexing and search. Setting the parameter to `true` also enables the display of the **Search Policies → Basic** screen, which uses free-text search. The `FreeTextSearchEnabled` parameter has no effect on the user interface unless you also set the `EnableDisplayBasicSearchTab` *script parameter* to `true`. Script parameters are defined initially through Studio but administered on the **Script Parameters** page of the **Administration** tab.

Default: `False`

See also

- “Script Parameters” on page 100
- “Free-text Search Configuration” on page 341

PolicySearchMaxResult

Number of maximum results to return from a database policy search. This parameter has no effect on free-text policy search.

Default: 300

See also

- “Search Overview” on page 329

Security Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to application security.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

EnableDownlinePermissions

If `UseACLPermissions` is `true`, then setting this parameter to `true` means that supervisors inherit permissions on an object that has been added for a supervised user or group.

Default: `True`

FailedAttemptsBeforeLockout

Number of failed attempts that PolicyCenter permits before locking out a user. For example, setting this value to 3 means that the third unsuccessful try locks the account from further repeated attempts. This integer value must be 1 or greater. A value of -1 disables this feature.

Default: 3

Minimum: -1

LockoutPeriod

Time in seconds that PolicyCenter locks a user account. A value of -1 indicates that a system administrator must manually unlock a locked account.

Default: -1

LoginRetryDelay

Time in milliseconds before a user can retry after an unsuccessful login attempt. This integer value must be zero (0) or greater.

Default: 0

Minimum: 0

MaxPasswordLength

New passwords must be no more than this many characters long. This integer value must be zero (0) or greater.

Default: 16

MinPasswordLength

New passwords must be at least this many characters long. For security purposes, Guidewire recommends that you set this value to 8 or greater. This integer value must be zero (0) or greater. If 0, then Guidewire PolicyCenter does not require a password. (Guidewire does not recommend this.)

Default: 8

Minimum: 0

RestrictContactPotentialMatchToPermittedItems

Whether PolicyCenter restricts the match results from a contact search screen to those that the user has permission to view.

Default: True

RestrictSearchesToPermittedItems

Whether PolicyCenter restricts the results of a search to those that the user has permission to view.

Default: True

SessionTimeoutSecs

Use to set the session expiration timeout, in seconds. By default, a session expires after three hours ($60 * 60 * 3 = 10800$ seconds).

- The minimum value to which you can set this parameter is five minutes ($60 * 5 = 300$ seconds).
- The maximum value to which you can set this parameter is one week ($3600 * 24 * 7 = 604800$ seconds)

Default: 10800

Minimum: 300

Maximum: 604800

Side-by-Side Quoting Parameters

Guidewire provides the following configuration parameters in `config.xml` that relate to side-by-side quoting.

PolicyCenter provides a number of configuration parameters related to the display of side-by-side quotes in the PolicyCenter interface. For a discussion of side-by-side quoting, see “Side-by-side Quoting” on page 161 in the *Application Guide*.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

RenewalMaxSideBySideQuotes

Sets an upper limit on the number of versions to show on a side-by-side quote page for a policy renewal.

Default: 4

SideBySide

Determines whether validation warnings block **Quote All** in side-by-side quoting.

In the default configuration, **Quote All** in the **Side by Side Quoting** screen does not produce quotes when there are validation warnings. This is to prevent PolicyCenter from generating invalid quotes. Guidewire expects that the default implementation is suitable for most implementations.

Default: true

See also

- For more information about setting this parameter, see “Configuring Quote All to Ignore Validation Warnings” on page 654.

SubmissionMaxSideBySideQuotes

Sets an upper limit on the number of versions to show on a side-by-side quote page for a policy submission.

Default: 4

PolicyChangeMaxSideBySideQuotes

Sets an upper limit on the number of versions to show on a side-by-side quote page for a policy change.

Default: 4

User Interface Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to the PolicyCenter interface.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

ActionsShortcut

The keyboard shortcut to use for the **Actions** button.

Default: A

AutoCompleteLimit

The maximum number of autocomplete suggestions to show.

Default: 10

InputMaskPlaceholderCharacter

The character to use as a placeholder in masked input fields.

Default: . (period)

ListViewPageSizeDefault

The default number of entries that PolicyCenter displays in each page in a list view, if the page does not explicitly specify this value. This integer value must be at least 1.

Default: 15

Minimum: 1

MaxBrowserHistoryItems (Obsolete)

This parameter is obsolete. Do not use it.

QuickJumpShortcut

The keyboard shortcut to use to activate the QuickJump box.

Default: / (forward slash)

UISkin

Name of the PolicyCenter interface skin to use.

Default: Titanium

WizardNextShortcut

Keyboard shortcut for the **Next** button in the set of wizard buttons. This value can be `null`.

WizardPrevShortcut

Keyboard shortcut for the **Previous** button in the set of wizard buttons. This value can be `null`.

WizardPrevNextButtonsVisible

Controls the visibility of the **Previous** and **Next** buttons in a wizard. If set to `true`, PolicyCenter renders the **Back** button on the first wizard step grayed-out to indicate that it is not available. A value of `null` is acceptable.

Default: False

Work Queue Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to the work queue.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 36.

InstrumentedWorkerInfoPurgeDaysOld

Number of days to retain instrumentation information for a worker instance before PolicyCenter deletes it.

Default: 45

WorkItemCreationBatchSize

The maximum number of work items for a work queue writer to create for each transaction.

Default: 100

WorkItemPriorityMultiplierSecs

Used to calculate the `AvailableSince` field for new work items. For new work items without a priority, PolicyCenter sets `AvailableSince` to the current time. Later, PolicyCenter checks out work items from the work queue in ascending order by `AvailableSince`, so work items without a priority are checked out in the order they were created.

You can assign a priority to new work items by implementing the Work Item Priority plugin (`IWorkItemPriorityPlugin`). For new work items with a priority, PolicyCenter sets `AvailableSince` according to the following formula:

```
workItem.AvailableSince = CurrentTime - (workItem.Priority * WorkItemPriorityMultiplierSecs)
```

Work items with higher priorities have earlier `AvailableSince` values than work items with lower priorities. Therefore, work items with higher priorities are checked out before ones with lower priorities because their `AvailableSince` values are earlier.

Priority influences the calculation of AvailableSince only at the time work items are created. If a worker throws an exception while processing a work item, PolicyCenter reverts the status of the work item from checkedout to available. At the same time, PolicyCenter resets AvailableSince according to the following formula:

```
workItem.AvailableSince = CurrentTime + RetryInterval
```

Work items are retried in the order they encounter exceptions, irrespective of priority.

IMPORTANT Prioritization affects only work items of type `WorkflowWorkItem` or its derivatives.

Default: 600

WorkItemRetryLimit

The maximum number of times that PolicyCenter retries an orphaned or failed work item.

Guidewire logs a `ConcurrentDataChangeException` generated by workers at different levels depending on context. If the `ConcurrentDataChangeException` occurs on processing the work items, PolicyCenter logs the error only if the number of attempts exceeds the configured value of the `WorkItemRetryLimit`. Otherwise, PolicyCenter logs the debug message instead.

Default: 3

WorkQueueHistoryMaxDownload

The maximum number of `ProcessHistory` entries to consider when producing the Work Queue History download. The valid range is from 1 to 525600. (The maximum of 525,600 is $60*24*365$, which is one writer running every minute for a year.)

Default: 10000

WorkQueueThreadPoolMaxSize

Maximum number of threads in the work queue thread pool. This must be greater than or equal to `WorkQueueThreadPoolMinSize`.

Default: 50

Set For Environment: Yes

WorkQueueThreadPoolMinSize

Minimum number of core threads in the work queue thread pool.

Default: 0

Set For Environment: Yes

WorkQueueThreadsKeepAliveTime

Keep alive timeout (in seconds) for additional on-demand threads in the work queue thread pool. An additional on-demand thread is terminated if it is idle for more than the time specified by this parameter.

Default: 60

Set For Environment: Yes

The Guidewire Development Environment

Working with Guidewire Studio

This topic describes Guidewire Studio and the Studio development environment.

This topic includes:

- “What Is Guidewire Studio?” on page 83
- “Using Studio with IntelliJ IDEA Ultimate Edition” on page 84
- “Starting Guidewire Studio” on page 84
- “The Studio Development Environment” on page 85
- “Improving Studio Performance” on page 86
- “Working with the QuickStart Development Server” on page 87
- “PolicyCenter Configuration Files” on page 89
- “Setting Font Display Options” on page 90

What Is Guidewire Studio?

Guidewire Studio is the IDE (integrated development environment) for creating and managing PolicyCenter application resources. These resources include Gosu rules, classes, enhancements and plugins, and all configuration files used by PolicyCenter to build and render the application.

Guidewire Studio is based upon IntelliJ IDEA Community Edition, a powerful and popular IDE.

Using Guidewire Studio, you can:

- Create and edit individual rules, and manage these rules and their order of consideration within a rule set
- Create and manage PCF pages, workflows, entity names, and display keys
- Create and manage Gosu classes and entity enhancements
- Create and manage the PolicyCenter data entities, business objects, and data types
- Manage plugins and message destinations
- Configure database connections

- Manage the PolicyCenter product model

IMPORTANT Do not create installation directories that have spaces in the name. This can prevent Guidewire Studio from functioning properly.

Using Studio with IntelliJ IDEA Ultimate Edition

Guidewire Studio is bundled with the Community Edition of IntelliJ IDEA, a free version of this popular IDE. If desired, you can configure Studio to work with the Ultimate Edition of IntelliJ IDEA instead. To use the Ultimate Edition, you must obtain your own license for it from IntelliJ.

To configure Guidewire Studio to use IntelliJ IDEA Ultimate Edition

1. In your PolicyCenter installation directory, create a text file named `studio.ultimate` that contains the full path of your IntelliJ IDEA Ultimate Edition installation directory. For example:

```
C:\Program Files (x86)\JetBrains\IntelliJ IDEA 12.1.7.
```

2. Run Guidewire Studio.
3. When prompted for your IntelliJ IDEA Ultimate Edition license, provide it.

Starting Guidewire Studio

You start Studio from the command line.

To start Guidewire Studio

4. Do either of the following:

- Open a command window and navigate to the application root directory. At the command prompt, type:
`studio`
- Open a command window and navigate to the application `bin` directory. At the command prompt, type:
`gwpc studio`

The first time that you start Guidewire Studio, it may take some extra time to load and index configuration data. Subsequent starts, however, generally load much more quickly.

To stop Guidewire Studio

To stop Guidewire Studio, select **Exit** from the Studio **File** menu. It is also possible to stop Studio by closing its window (by clicking the **x** in the upper right-hand corner of the window).

Restarting Studio

Certain changes that you make in Studio require that you restart Studio before it recognizes those changes. For example, if you add a new workflow type, then you must stop and restart Studio before a Gosu class that you create recognizes the workflow.

Guidewire does not strictly require that you always stop and restart Studio after a data model change. However, it is one way to test that you have not inadvertently made a typing error, for example.

Note: If you modify the base configuration data model, you must start (or restart) the application server. This forces a database upgrade. See “Deploying Configuration Files” on page 31 for more information.

The Studio Development Environment

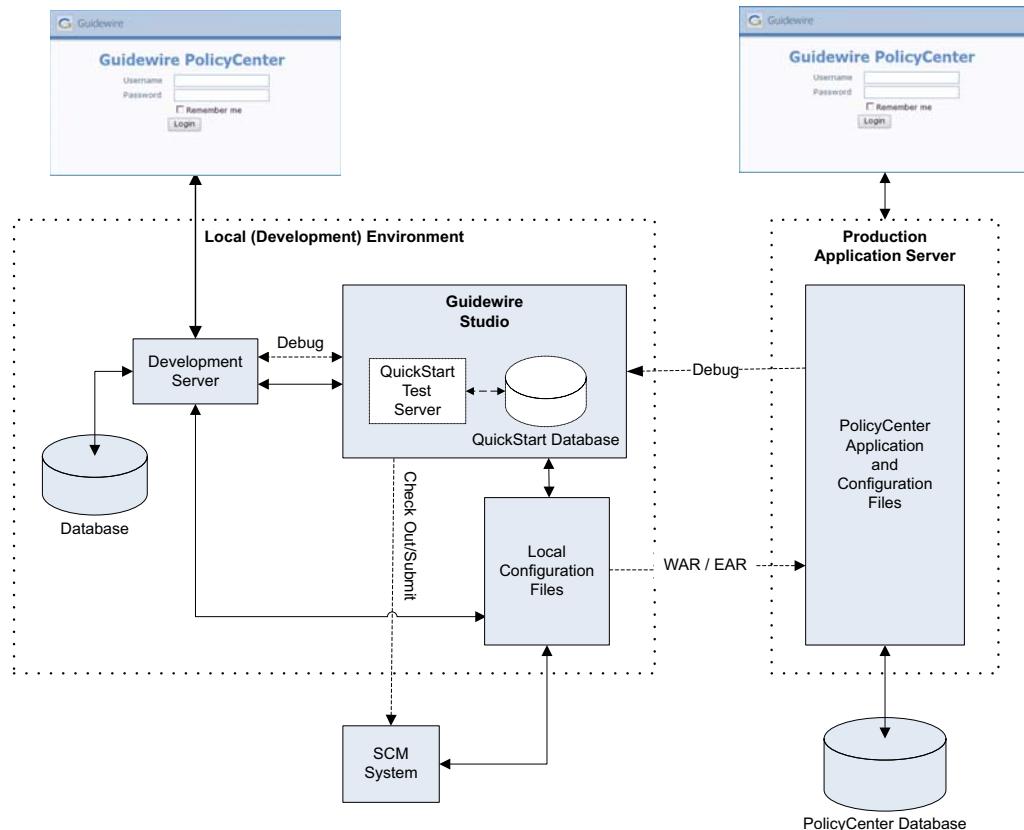
Guidewire Studio is a stand-alone development application that runs independently of Guidewire PolicyCenter. You use Studio to build and test application customization in a development or test mode before deploying your changes to a production server. Any changes that you make to application files through Studio do not automatically propagate into production. You must specifically build a .war file and deploy it to a server for the changes to take effect. (Studio and the production application server—by design—do not share the same configuration file system.)

Guidewire recommends that you not run Studio on a machine with an encrypted hard drive. If you run Guidewire Studio on a machine with hard drive encryption, Studio can take 15 or more seconds to refresh. This slow refresh can happen when you switch focus from the Studio window to something else, such as the browser, and back again.

To assist with this development and testing process, Guidewire bundles the following with the PolicyCenter application:

- A QuickStart development server
- A QuickStart database
- A QuickStart server used for testing that you cannot control
- A QuickStart database used for testing that is separate from the QuickStart development database

The following diagram illustrates the connections between Guidewire Studio, the bundled QuickStart applications, the local file system, and the PolicyCenter application server. You use the QuickStart test server and test database for testing only as PolicyCenter controls them internally. You can use either the bundled QuickStart development server bundled with Guidewire PolicyCenter or use an external server such as Tomcat. In general, dotted lines indicate actions on your part that you perform manually. For example, you must manually create a .war file and manually move it to the production server. The system does not do this for you.



Improving Studio Performance

There are a number of things you can do to improve performance of Studio.

- You can make code compile faster or not use up Studio memory. See “Improving Performance of Code Compilation” on page 86.
- You can provide overall improvement by providing Studio with more memory. See “Increasing Memory Available to Studio” on page 86.

Improving Performance of Code Compilation

There are a number of techniques for working with Gosu that make code compilation faster. Additionally, if you are noticing slow compiles, it might be that you are running out of memory and need to use the external compiler.

Gosu Hints for Improving Gosu Compilation Speed

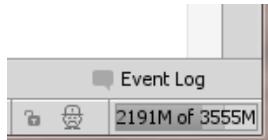
There are several things you can do with Gosu to improve compile speed:

- Write all your Gosu as case-sensitive code. See “Gosu Case Sensitivity” on page 95.
- Configure Gosu class preloading. See “Preloading Gosu Classes” on page 98.

Using the External Compiler

If you are using the techniques described in the previous topic and you are experiencing slow responses from Studio, the internal compiler might be using up too much available memory.

To see if memory is getting low, check your memory indicator in the bottom right-hand corner of the main Studio window:



If the amount of memory being used, on the left, is close to the total amount available, you can stop and restart Studio to see if more memory becomes available. After restarting Studio, if you see that there is less memory being used, then the internal compiler is using up memory.

You can avoid using up Studio memory by setting up Studio to use the external compiler. Each compile will be a little slower, but you will not use up all your memory over time.

To set Studio to use the external compiler

1. In Studio, navigate to **File** → **Settings** → **Guidewire Studio** and click **Guidewire Studio**.
2. In the **Guidewire Studio** settings window, find the settings for **External Compiler**.
3. Set **External Compile Threshold** to 0.
4. Restart Studio.

Increasing Memory Available to Studio

If Studio is running slowly, try the techniques for improving performance that are described in the previous topics. If it still runs slowly, you can increase the amount of memory available to Studio can make it respond more quickly.

In the base configuration, the amount of memory available to the JVM for Studio is:

- **Starting heap size (.xms)** – 2000 megabytes

- **Maximum heap size (.xmx)** – 4000 megabytes
- **Maximum permanent size (.maxperm)** – 500 megabytes

To change the Studio memory settings

1. In Studio, navigate in the Project window to configuration → etc and double-click `memory.properties`.
2. Set the following values to higher numbers as needed. The following settings are examples of increased values you might set:
 - a. `com.intellij.idea.Main.xms=2048`
 - b. `com.intellij.idea.Main.xmx=8192`
 - c. `com.intellij.idea.Main.maxperm=1024`
3. Save the file.
4. Restart Studio.

Working with the QuickStart Development Server

It is possible to use any of the supported application servers in a development environment, rather than the embedded QuickStart server. To do so, you need to point PolicyCenter to the configuration resources edited by Guidewire Studio. This requires additional configuration, described for each application server type in “Deploying PolicyCenter to the Application Server” on page 82 in the *Installation Guide*.

You cannot start the QuickStart development server directly from Studio. You cannot manually start the QuickStart test server because Studio manages it internally. Instead, you start this server from the command line. Use the following command to start the QuickStart server from the `bin` directory of your Studio installation

```
PolicyCenter/bin/gwpc dev-start
```

Use the following `dev` commands as you work with the QuickStart server. See “Commands Reference” on page 117 in the *Installation Guide* for a complete list of commands and how to use them.

Command	Action
<code>gwpc dev-start</code>	Starts the Development server.
<code>gwpc dev-stop</code>	Stops the Development server.
<code>gwpc dev-dropdb</code>	Resets QuickStart database associated with the QuickStart development server.

In each application configuration, Guidewire provides the following QuickStart default port settings:

Application	Port
ClaimCenter	8080
PolicyCenter	8180
ContactManager	8280
BillingCenter	8580

For more information on the `gwpc dev` commands, see “Installing the QuickStart Development Environment” on page 48 in the *Installation Guide*.

Connecting the Development Server to a Database

PolicyCenter running on the QuickStart development server can connect to the same kinds of databases as any of the other Guidewire-supported application servers. However, for performance reason, Guidewire recommends that you use the bundled QuickStart database. Guidewire optimizes this database for fast development use. It can run in either of the following modes:

Mode	Description
file mode	The database persists data to the hard drive (the local file system), which means that the data can live from one server start to another. This is the Guidewire-recommended default configuration.
memory mode	The database does not persist data to the hard drive and it effectively drops the database each time you restart the server. Guidewire does not recommend this configuration.

You set configuration parameters for the QuickStart database associated with the development server in `config.xml`. For example:

```
<!-- H2 (meant for development/quickstart use only!) -->
<database name="PolicyCenterDatabase" driver="dbcp" dbtype="h2" printcommands="false"
    autoupgrade="true" checker="false">
    <param name="jdbcURL" value="jdbc:h2:file:/tmp/guidewire/pc"/>
    <param name="stmtPool.enabled" value="false"/>
    <param name="maxWait" value="30000"/>
    <param name="CACHE_SIZE" value="32000"/>
</database>
```

To set the database mode

In the base configuration, the QuickStart database runs in *file mode*. You set the database mode using the `jdbcURL` parameter value. In file mode the `jdbcURL` parameter value points to an actual file location. For example:

```
<param name="jdbcURL" value="jdbc:h2:file:/tmp/guidewire/pc"/>
```

Guidewire uses `/tmp/guidewire/pc` as the file location in the base configuration.

To drop the QuickStart database

Occasionally, you may want (or need) to drop the QuickStart database. For example, Guidewire recommends that you drop the QuickStart database if you make changes to the PolicyCenter data model.

To drop the database, use the `gwpc dev-dropdb` command. To drop the database manually, delete the files from the directory specified by the `jdbcURL` parameter (by default, `<root>/tmp/guidewire/pc`). The server must be down if you delete the directory.

Deploying Your Configuration Changes

To deploy your configuration changes to an actual production server, you must build a `.war` file and deploy it on the application server. By design, you cannot directly deploy configuration files from Studio to the application server.

As the bundled QuickStart development server and Studio share the same configuration directory, you do not need to deploy your configuration changes to the QuickStart development server.

To hot-deploy PCF files

Editing and saving PCF files in the **Page Configuration (PCF)** editor does not automatically reload them in the QuickStart server, even if there is a connection between it and Studio. Instead, first save your files, then navigate to the PolicyCenter web interface on the deployment server. After you log into the interface, reload the PCF configuration using either the **Internal Tools** page or the **Alt+Shift+L** shortcut.

You can also reload display keys this way, as well.

You do not actually need to be connected to the server from Studio to reload PCF files.

PolicyCenter Configuration Files

WARNING Do not attempt to modify any files other than those in the `PolicyCenter/modules/` configuration directory. Any attempt to modify files outside of this directory can cause damage to the PolicyCenter application and prevent it from starting thereafter.

Installing Guidewire PolicyCenter creates the following directory structure:

Directory	Description
.idea	Contains configuration and settings for IntelliJ IDEA.
admin	Contains administrative tools. See "PolicyCenter Administrative Commands" on page 157 in the <i>System Administration Guide</i> for descriptions.
bin	Contains the gwpc batch file and shell script used to launch commands for building and deploying. See "Commands Reference" on page 117 in the <i>Installation Guide</i> .
build	Contains products of build commands such as exploded .war files and the data and security dictionaries. This directory is not present when you first install PolicyCenter. The directory is created when you run one of the build commands.
dist	Guidewire application .war, and .jar files are built in this directory. The directory is created when you run one of the build commands to generate .war files.
doc	HTML and PDFs of PolicyCenter documentation.
idea	Contains IntelliJ IDEA application files.
java-api	Contains the Java API libraries created by running the gwpc regen-java-api command. See "Regenerating Integration Libraries and WSDL" on page 31 in the <i>Integration Guide</i> .
logs	Contains log files.
modules	Contains subdirectories including configuration resources for each application component.
repository	Contains necessary PolicyCenter files.
soap-api	Contains the web service WSDL files that the gwpc regen-soap-api command generates. See "Regenerating Integration Libraries and WSDL" on page 31 in the <i>Integration Guide</i> .
solr	For internal use only.
studio	Contains Studio preferences and TypeInfo database caches. Studio generates this directory when you first launch Studio.
template	Contains template files.
webapps	Contains necessary files for use by the application server.

Edited Resource Files Reside in the Configuration Module Only

The configuration module is the only place for configured resources. As PolicyCenter starts, a checksum process verifies that no files have been changed in any directory except for those in the configuration directory. If this process detects an invalid checksum, PolicyCenter does not start. In this case, overwrite any changes to all modules except for the configuration directory and try again.

Guidewire recommends that you use Studio to edit configuration files to minimize the risk of accidentally editing a file outside the configuration module.

Key Directories

The installation process creates a configuration environment for PolicyCenter. In this environment, you can find all of the files needed to configure PolicyCenter in two directories:

- The main directory of the configuration environment. In the default PolicyCenter installation, the location of this directory is `PolicyCenter/modules/configuration`.
- `PolicyCenter/modules/configuration/config` contains the application server configuration files.

The installation process also installs a set of system administration tools in `PolicyCenter/admin/bin`.

PolicyCenter runs within a J2EE server container. To deploy PolicyCenter, you build an application file suitable for your server and place the file in the server's deployment directory. The type of application file and the deployment directory location is specific to the application server type. For example, for PolicyCenter (deployed as the `pc.war` application) running on a Tomcat J2EE server on Windows, the deployment directory might be `C:\Tomcat\webapps\pc`.

Setting Font Display Options

To set how Studio handles various font and Gosu editor options, do the following. First, navigate to **File → Settings → Editor → Colors & Fonts**. Use the **Colors & Fonts** menu selections to set Studio display of text in the editors. For example, if you click **Gosu**, you can set the font type and size of Gosu code in the editor. You can also set how Studio displays specific Gosu code items, such as keywords or operators. Studio displays a code sample at the bottom of the dialog that reflects your settings.

Using this menu, you can set values for:

- Text font and size
- Character format properties
- Foreground and background colors of various language elements

Note: You can set anti-aliasing of fonts in the editor **Appearance** settings page at **File → Settings → Editor → Appearance**. Additionally, you can set font size zooming with the **Ctrl+Mouse wheel** in the main **Editor** settings page at **File → Settings → Editor**.

You can configure Studio to open XML files directly in an XML editor that is external to Guidewire Studio. To facilitate XML editors, XML documents provide the `xmlns` attribute to specify a URL to an XSD file. An XSD file defines the *namespace* for elements and attributes in the XML document. The XSD file provides information that lets the XML editor validate the correctness of XML documents.

Note: The `xmlns` attribute currently is optional. However, Guidewire strongly recommends that you add the attribute to your entity and typelist files, because Guidewire reserves the right to make this attribute required in the future.

Namespace URLs

Use the appropriate namespace URL for each type of metadata file:

- **Entity files** – Use the following for entity definition files (`.eti` and `.etx`):
`<entity xmlns="http://guidewire.com/datamodel" ...`
- **Typelist files** – Use the following for typelist definition files (`.tti` and `.ttx`):
`<typelist xmlns="http://guidewire.com/typelists" ...`

Configuring External XML Editors

You can configure many XML editors to associate namespaces with XSDs. However, merely defining the namespace within Guidewire PolicyCenter is not sufficient to inform the XML editor which XSD to use to validate an XML document. You must configure your external XML editor manually to associate namespaces with the XSDs.

IMPORTANT If you use a third-party tool to edit PolicyCenter configuration files, Guidewire recommends that you work with one that fully supports UTF-8 file encoding. Editing tools that do not handle UTF-8 characters correctly can create errors in PolicyCenter. For XML files, you can use a different encoding, as long as you specify the encoding in the XML prolog. For all other files other than XML, use UTF-8.

PolicyCenter Studio and Gosu

This topic discusses how to work with Gosu code in PolicyCenter Studio.

This topic includes:

- “Studio and the DCE VM” on page 93
- “Gosu Building Blocks” on page 94
- “Gosu Case Sensitivity” on page 95
- “Working with Gosu in PolicyCenter Studio” on page 95
- “Gosu Packages” on page 95
- “Gosu Classes” on page 96
- “Gosu Enhancements” on page 99
- “The Guidewire XML Model” on page 100
- “Script Parameters” on page 100

Studio and the DCE VM

The PolicyCenter application server and Guidewire Studio require a JVM (Java Virtual Machine). The version of the JVM depends on the servlet container and operating system on which the application server runs.

Guidewire strongly recommends the use of the DCE VM for development in the QuickStart environment. Guidewire does not support the DCE VM for other application servers or in a production environment.

The Dynamic Code Evolution Virtual Machine (DCE VM) is a modified version of the Java HotSpot Virtual Machine (VM). The DCE VM supports any redefinition of loaded classes at runtime. You can add and remove fields and methods and make changes to the super types of a class using the DCE VM. The DCE VM is an improvement to the HotSpot VM, which only supports updates to method bodies.

DCE VM Limitations

If you reload Gosu classes using hotswap on the DCEVM, it is possible to add new static fields (again, only on the DCE VM). However, Gosu does not execute any initializers for those static variables. For example, if you add the following static field to a class:

```
public static final var NAME = "test"
```

Gosu adds the NAME field to the class dynamically. However, the value of the field is `null` until you restart the server (or Studio, if you are running the code from the Studio Gosu Tester). If you need to initialize a newly added static field, you must write a static method that sets the variable and then executes that code.

For example, suppose that you added the following static method to class `MyClass`:

```
public static var x : int = 10
```

To initialize this field, write code to set the static variable to the value that you expect and then execute that code:

```
MyClass.x = 10
```

This does not work if the field is `final`.

Note: Adding an instance variable rather than a static variable with an initializer also results in `null` values on existing instances of the object. However, any newly-constructed instances of the object will have the field initialized.

See also

- For details on how to select the proper JVM for your installation, see “Installing Java” on page 43 in the *Installation Guide*.
- “Installing the Dynamic Code Evolution Virtual Machine” on page 44 in the *Installation Guide*.

Gosu Building Blocks

Guidewire provides a number of building blocks to assist you in implementing, configuring, and testing your business logic in PolicyCenter. These include the following:

- Gosu classes and enhancements
- Gosu base library methods
- Gosu rules
- Gosu tests
- Gosu script parameters

For information on each of these, see the following:

- For general information on Gosu classes, see “Classes” on page 191 in the *Gosu Reference Guide*.
- For information on the PolicyCenter base configuration classes see, “PolicyCenter Base Configuration Classes” on page 96.
- For information on the `@export` annotation and how it affects a class in Studio, see “Class Visibility in Studio” on page 98.
- For general information on Gosu enhancements, see “Enhancements” on page 229 in the *Gosu Reference Guide*.
- For information on using Gosu business rules within Guidewire PolicyCenter, see “Rules Overview” on page 15 in the *Rules Guide*.
- For information on script parameters and how to use them in Gosu code, see “Script Parameters” on page 100.

Gosu Case Sensitivity

Gosu is case-sensitive for most types. For example, if a type is declared as `MyClass`, you cannot refer to it as `myClass` or `myclass`.

All Guidewire entity types are case-insensitive. However, it is recommended that code maintains a case-sensitive approach. Doing so ensures that your code compiles and runs more quickly than if case-sensitivity is ignored.

Standard conventions exist for the capitalization of different language elements. The following table lists the conventions.

Language element	Standard capitalization	Example
Gosu keywords	Always specify Gosu keywords correctly as they are declared, typically lowercase. Java keywords are case-sensitive.	<code>if</code>
Type names, including class names	Uppercase first character	<code>DateUtil</code> <code>Claim</code>
Local variable names	Lowercase first character	<code>myClaim</code>
Property names	Uppercase first character	<code>CarColor</code>
Method names	Lowercase first character	<code>printReport</code>
Package names	Lowercase all letters in packages and subpackages	<code>com.mycompany.*</code>
Java types (case sensitive)	Java types require case sensitivity Always specify Java types correctly as they are declared. Java type names are case-sensitive.	<code>java.util.String</code>

Guidewire strongly recommends that all code follows a case-sensitive approach. To assist you, Studio highlights issues with case sensitivity. It also provides a tool to automatically fix all case sensitivity issues so your code compiles and runs as fast as possible.

Working with Gosu in PolicyCenter Studio

It is possible to create the following by selecting **New** from the **Classes** contextual right-click menu:

Classes → New →	For information, see...
Class	<ul style="list-style-type: none">“Classes” on page 191 in the <i>Gosu Reference Guide</i>“Gosu Classes” on page 96
Interface	<ul style="list-style-type: none">“Interfaces” on page 213 in the <i>Gosu Reference Guide</i>
Enhancement	<ul style="list-style-type: none">“Enhancements” on page 229 in the <i>Gosu Reference Guide</i>“Gosu Enhancements” on page 99
Template	<ul style="list-style-type: none">“Gosu Templates” on page 351 in the <i>Gosu Reference Guide</i>
Package	<ul style="list-style-type: none">“Gosu Packages” on page 95
GX Model	<ul style="list-style-type: none">“The Guidewire XML (GX) Modeler” on page 304 in the <i>Gosu Reference Guide</i>

Gosu Packages

Guidewire PolicyCenter stores Gosu classes, enhancements, and templates in hierarchical structure known as packages. To access a package, expand the **Classes** node in the Studio Resources tree.

To create a new package

It is possible to nest package names to create a dot-separated package name by selecting a package and repeating these steps.

1. Select **Classes** in the **Resources** tree.
2. Right-click, select **New**, then **Package** from the menu.
3. Enter the name for this package.
4. Click **OK** to save your work and exit this dialog.

Note: You can only delete an empty package.

Gosu Classes

Gosu classes correspond to Java classes. Gosu classes reside in a file-based package structure. You can extend classes in the base configuration of PolicyCenter to add properties and methods, and you can write your own Gosu classes. You define classes in Gosu, and you access the properties and call the methods of Gosu classes from Gosu code within methods.

You create and reference Gosu classes by name, just as in Java. For example, you define a class named `MyClass` in a package named `MyPackage`. You define a method on your class named `getName`. After you define your class, you can instantiate an instance of the class and call the method on that instance, as the following Gosu sample code demonstrates.

```
var myClassInstance = new MyPackage.MyClass()  
var name = myClassInstance.getName()
```

Studio stores enhancement files in the **Classes** folder in the **Resources** tree. Gosu class files end in `.gs`.

To create a new class

1. First create a package for your new class, if you have not already done so.
2. Select the package in the **configuration** tree.
3. Right-click, select **New**, then **Gosu Class** from the menu.
4. Enter the name for this class. (You can also set the resource context for this class at this time.)
5. Click **OK** to save your work and exit this dialog.

See also

- “PolicyCenter Base Configuration Classes” on page 96
- “Class Visibility in Studio” on page 98
- “Preloading Gosu Classes” on page 98
- “Classes” on page 191 in the *Gosu Reference Guide*

PolicyCenter Base Configuration Classes

The **Classes** resource folder contains Guidewire classes and enhancements—divided into packages—that provide additional business functionality. In the base configuration, Studio contains the following packages in the **Classes** folder:

- `com`
- `gw`
- `wsi`

If you create new classes and enhancements, Guidewire recommends that you create your own subpackages in the **Classes** folder, rather than adding to the existing Guidewire folders.

The com Package

In the base configuration, the `com` package contains a single Gosu class:

```
com.guidewire.pl.quickjump.BaseCommand
```

For a discussion of the QuickJump functionality, see “[Implementing QuickJump Commands](#)” on page 119.

The gw Package

In the base PolicyCenter configuration, the `gw.*` Gosu class libraries contain a number of Gosu classes, divided into subpackages by functional area. To access these libraries, you merely need to type `gw.` (gw dot) in a Studio editor. The following subpackages under the `gw` package play an important role in Studio:

- `gw.api.*`
- `gw.plugin`

gw.api.*

There are actually two `gw.api` packages that you can access:

- One consists of a set of built-in library functions that you can access and use, but not modify.
- The other set of library functions is visible in the Studio **Classes** folder in the configuration tree. You can not only access these classes but also modify them to suit your business needs.

You access both the same way, by entering `gw.api` in the Gosu editor. You can then choose a package or class that falls into one category or the other. For example, if you enter `gw.api.` in the Gosu editor, the Studio **Complete Code** feature provides you with the following list:

- `activity`
- `address`
- `admin`
- `...`

In this case, the `activity` and `admin` packages contain read-only classes. The `address` package is visible in Studio, in the **Classes** folder.

gw.plugin

If you create a new Gosu plugin, place your plugin class in the `gw.plugin` package.

- For information on how to use Studio to work with plugins, see “[Using the Plugins Registry Editor](#)” on page 109.
- For information on various types of plugins and how to implement plugins, see “[Plugin Overview](#)” on page 123 in the *Integration Guide*.

The wsi Package

PolicyCenter provides a fully WS-I standard-compliant web services layer for both server (publishing) and client (consuming) web service APIs. The `wsi` package provides means of working with WS-I compliant web services.

See also

- “[Web Services Introduction](#)” on page 37 in the *Integration Guide*.
- “[Reference of All Built-in Web Services](#)” on page 39 in the *Integration Guide*.
- “[Calling Web Services from Gosu](#)” on page 75 in the *Integration Guide*.
- “[Using the Web Service Editor](#)” on page 116.

Class Visibility in Studio

For a Guidewire-provided Gosu class to be directly visible Studio, Guidewire must mark that class with the @export annotation. Thus, it is possible to view a class file in the application file structure, but to not be able to view or access the file in the Studio Gosu editor. This is because the class file is missing the @export annotation.

If you need to access the class, simply create a new class and have it extend or subclass the class that you need. For example, in PolicyCenter, the application source code defines a CCPCSearchCriteria class. This class is visible in the application file structure as a read-only file in the following location:

```
PolicyCenter/modules/configuration/gsrc/gw/webservice/pc/pc700/ccintegration/ccentitie
```

To access the class functionality, first create a new class in the following Studio Classes package:

```
gw.webservice.pc.ccintegration.v2.ccentities
```

You then have this class extend CCPCSearchCriteria, for example:

```
package gw.webservice.pc.ccintegration.v2.ccentities

uses java.util.Date
uses gw.api.web.product.ProducerCodePickerUtil
uses gw.api.web.producer.ProducerUtil

class MyClass extends CCPCSearchCriteria {
    var _accountNumber : String as AccountNumber
    var _asOfDate : Date as AsOfDate
    var _nonRenewalCode : String as NonRenewalCode
    var _policyNumber : String as PolicyNumber
    var _policyStatus : String as PolicyStatus
    var _producerCodeString : String as ProducerCodeString
    var _producerString : String as ProducerString
    var _product : String as Product
    var _productCode : String as ProductCode
    var _state : String as State
    var _firstName : String as FirstName
    var _lastName : String as LastName
    var _companyName : String as CompanyName
    var _taxID : String as TaxID

    construct() { }

    override function extractInternalCriteria() : PolicySearchCriteria {
        var criteria = new PolicySearchCriteria()
        criteria.SearchObjectType = SearchObjectType.TC_POLICY
        ...
    }
}
```

Preloading Gosu Classes

PolicyCenter provides a preload mechanism to support pre-compilation of Gosu classes, as well as other primary classes. The intent is to make the system more responsive the first time requests are made for that class. This is meant to improve application performance by preloading some of the necessary application types.

To support this, Guidewire provides a `preload.txt` file in **Other Resources** to which you can add the following:

Static method invocations	<p>Static method invocations dictate some kind of action and have the following syntax:</p> <pre>type#method</pre> <p>The referenced method must be a static, no-argument method. However, the method can be on either a Java or Gosu type.</p> <p>In the base configuration, Guidewire includes some actions on the <code>gw.api.startup.PreloadActions</code> class. For example, to cause all Gosu types to be loaded from disk, use the following:</p> <pre>gw.api.startup.PreloadActions#headerCompileAllGosuClasses</pre> <p>It is possible to add in your own static methods to use in this fashion as meets your business needs.</p>
Type names	<p>Type names can be either Gosu or Java types. You must use the fully-qualified name of the type. For any Java or Gosu type that you list in this file:</p> <ul style="list-style-type: none">Java – PolicyCenter loads the associated Java class file.Gosu – PolicyCenter parses and compiles the type down to byte-code, along with all blocks and inner classes of that type.

Note: Guidewire also provides a logging category, `Server.Preload`, that provides DEBUG level logging of all actions during server preloading of Gosu classes.

Populating the List of Types

To populate the list of types, Guidewire recommends that you first perform whatever actions you need to within PolicyCenter interface. Then, navigate to the **Loaded Gosu Classes** page (**Server Tools** → **Info Pages**) and copy and paste the list that you see there into the `preload.txt` file. The next time that you start the application server, PolicyCenter compiles those types on start-up.

Gosu Enhancements

Gosu enhancements provide additional methods (functionality) on a Guidewire entity. For example, suppose that you create an enhancement to the `Activity` entity. Within this enhancement, you add methods that support new functionality. Then, if you type `Activity.` (Activity dot) within any Gosu code, Studio automatically uses code completion on the `Activity` entity. It also automatically displays any methods that you have defined in your `Activity` enhancement, along with the native `Activity` entity methods.

Studio stores enhancement files in the `Classes` folder in the `Resources` tree.

- Gosu class files end in `.gs`.
- Gosu enhancement files end in `.gsx`.

The Gosu language defines the following terms:

- **Classes** – Gosu classes encapsulate data and code for a specific purpose. You can subclass and extend existing classes. You can store and access data and methods on an instance of the class or on the class itself. Gosu classes can also implement Gosu interfaces.
- **Enhancements** – Gosu enhancements are a Gosu language feature that allows you to augment classes and other types with additional concrete methods and properties. For example, use enhancements to define additional utility methods on a Java class or interface that you cannot directly modify. Also, you can use an enhancement to extend Gosu classes, Guidewire entities, or Java classes with additional behaviors.

To create a new enhancement

1. First create a package for your new class, if you have not already done so.
2. Select the package in the configuration tree.

3. Right-click, select **New**, then **Enhancement** from the menu.
4. Enter the name for this enhancement. Guidewire recommends strongly that you end each enhancement name with the word **Enhancement**. For example, if you create an enhancement for an **Activity** entity, name your enhancement **ActivityEnhancement**.
5. Enter the entity type to enhance. For example, if enhancing an **Activity** entity, enter **Activity**.
6. Click **OK** to save your work and exit this dialog.

See also

- “**Classes**” on page 191 in the *Gosu Reference Guide*
- “**Enhancements**” on page 229 in the *Gosu Reference Guide*

The Guidewire XML Model

It is possible to export business data entities, Gosu class data, and other types to a standard Guidewire XML format. It is also possible to select which properties to map in your XML model. By specifying what to map, PolicyCenter creates an XSD to describe XML that conforms to your XML model. At run time, you can export XML for this type and optionally choose to export only data model fields that changed. If you have more than one integration point that uses a type, you can create different XML models for each type.

In general, to create a new XML model, you do the following:

1. Navigate to the **Classes** package in which you want to create the XML model.
2. Right-click the package name and from the contextual menu, select **New → GX Model**.

See also

- For detailed information on the Guidewire XML model and the Guidewire XML Modeler in Studio, see “The Guidewire XML (GX) Modeler” on page 304 in the *Gosu Reference Guide*.

Script Parameters

Script parameters are Studio-defined resources that you can use as global variables within Gosu code. System administrators change their values on the **Administration** tab. Changes to values take effect immediately in Gosu code.

This topic includes:

- “**Script Parameters Overview**” on page 100
- “**Working with Script Parameters**” on page 101
- “**Referencing a Script Parameter in Gosu**” on page 102
- “**PolicyCenter Script Parameters**” on page 102

Script Parameters Overview

PolicyCenter uses the `ScriptParameters.xml` configuration file as the system of record for script parameter definitions and their initial values. You can create script parameters only from within Studio, by navigating to **configuration → config → resources → ScriptParameters.xml**. At the time of creation, Studio adds new script parameters to the `ScriptParameters.xml` configuration file. After creation, you manage the values of script parameters through the PolicyCenter user interface, not through Studio.

On server startup, PolicyCenter compares the list of script parameters that currently reside in the database to those in the `ScriptParameters` file. During the comparison, PolicyCenter does one of the following:

- **New script parameters** – PolicyCenter adds to the database new script parameters in the XML file that are not in the database. PolicyCenter propagates the initial values as set in the XML file to the database.
- **Existing script parameters** – PolicyCenter ignores existing script parameters in the XML file that already are in the database. PolicyCenter does not propagate changed values for existing parameters from the XML file to the database.

After a script parameter resides in the database, you manage it solely from the **Script Parameters** administration screen in the PolicyCenter administrative interface. You access the **Script Parameters** screen by first logging on with an administrative account, then navigating to **Administration** → **Script Parameters**.

IMPORTANT After you create a script parameter in Studio, PolicyCenter ignores subsequent changes that you make to the parameter value. You must make all subsequent changes to parameter values in the **Script Parameters** administration screen of the PolicyCenter user interface.

Script Parameters as Global Variables

There are several reasons to create global variables:

- You want a variable that is global in scope across the application that you can change or reset through the application interface.
- You want a variable to hold a value that you can use in any Gosu expression, and you want to change that value without editing the expression.

These two reasons for use of script parameters, while seemingly related, are entirely independent of each other.

- Use script parameters to create variables that you can change or reset through the PolicyCenter interface.
- Use Gosu class variables to create variables for use in Gosu expressions.

For information on Gosu class variables, see “*Gosu Classes and Properties*” on page 20 in the *Gosu Reference Guide*.

Script Parameter Examples

Suppose, for example, that you have exception rules that trigger when an activity is overdue for more than five (5) days. If you included the value “5” in all of the rules, you would have to modify the rules if you decided to change the value to ten (10). Instead, define a script parameter, set its value to five (5), and then use this parameter in the rules. To change the activity exception behavior, you need change only the parameter, and the rules automatically uses the new value.

More complex examples include:

- **Setting the number of days before the policy end date to start the renewal process** – It is possible that this varies by policy type and Line of Business.
- **Setting the maximum number of years for automatic policy renewal without an underwriting review** – After the set number of threshold years passes, the Rule engine generates an activity to manually review the policy.
- **Setting the date in which new policy terms come into effect** – This effectively requires all policies created after a given date to contain certain forms and amendment, for example.

Note: Script parameters are read-only within Gosu. You cannot set the value of a script parameter in a Gosu statement or expression.

Working with Script Parameters

In working with script parameters:

- You create script parameters and set their initial values in Guidewire Studio.
- You administer script parameters and modify their values in the PolicyCenter interface, on the **Administration** tab.

The application server references only the initial values for script parameters that you set in Guidewire Studio. Thereafter, the application server references the values that you set through the PolicyCenter interface and ignores subsequent changes to values that you set as set in Studio.

IMPORTANT After you create a script parameter in Studio, PolicyCenter ignores subsequent changes that you make to the parameter value. You must make all subsequent changes to parameter values in the **Script Parameters** administration screen of the PolicyCenter user interface.

To create a script parameter

1. In the Studio Project window, navigate to **configuration** → **config** → **resources** → **ScriptParameters.xml**.
2. Edit the XML and define your new parameter using the existing format as a guide.

To delete a script parameter

You can delete a script parameter if you no longer reference it in any Gosu expression.

1. In the Studio Project window, navigate to navigating to **configuration** → **config** → **resources** → **ScriptParameters.xml**.
2. Edit the XML and remove the element defining the parameter to delete.

Referencing a Script Parameter in Gosu

You can access a script parameter in Gosu through the globally accessible **ScriptParameters** object. Within Gosu, you access a parameter by using **ScriptParameters.paramname**.

For example, the following Gosu code determines if it is more than five days past an activities due date:

```
gw.api.util.DateUtil.daysSince( Activity.EndDate ) > 5
```

You can, instead, create a script parameter named **escalationTime**, set its value to 5, and rewrite the line as follows:

```
gw.api.util.DateUtil.daysSince( Activity.EndDate ) > ScriptParameters.escalationTime
```

Note: Guidewire recommends that you use Gosu class variables instead of script parameters to reference values in Gosu expressions. The exception would be if you needed the ability to reset the value from the PolicyCenter interface.

PolicyCenter Script Parameters

The default configuration of PolicyCenter includes the following script parameters. You administer script parameters and modify their values on the **Script Parameters** page of the **Administration** tab.

EnableDisplayBasicSearchTab

Whether to enable the free-text policy search user interface. Whenever this script parameter is set to **true**, the **Search Policies** screen displays tabs labeled **Basic** and **Advanced**. The **Basic** tab contains the user interface for free-text search. The **Advanced** screen contains the user interface for database search.

Whenever this script parameter is set to **false**, the **Search Policies** screen displays the **Advanced** screen only. Setting the **EnableDisplayBasicSearchTab** script parameter to **true** has no effect if the **FreeTextSearchEnabled** configuration parameter in **config.xml** is set to **false**.

Whenever a systems administrator runs the free-text batch load command while PolicyCenter is available to users, you must set this script parameter to **false** to hide the **Basic** search screen. Users continue to use the **Advanced** search screen without interruption. After the free-text batch load command finishes, set the **EnableDisplayBasicSearchTab** script parameter to **true** to restore the **Basic** search screen for users.

Changes to the value of the `EnableDisplayBasicSearchTab` script parameter take effect immediately. You do not need to restart the server.

Default: true

See also

- “Search Parameters” on page 75
- “Free-text Search Configuration” on page 341
- “Free-text Batch Load Command” on page 173 in the *System Administration Guide*

part III

Guidewire Studio Editors

Using the Studio Editors

This topic discusses the various editors available to you in Guidewire Studio.

This topic includes:

- “Editing in Guidewire Studio” on page 107
- “Working in the Gosu Editor” on page 108
- “Using Product Designer to Edit the Product Model” on page 108
- “Editing in Guidewire Studio” on page 107
- “Working in the Gosu Editor” on page 108
- “Using Product Designer to Edit the Product Model” on page 108

Editing in Guidewire Studio

Guidewire Studio displays PolicyCenter resources in the left-most Studio pane. After you select a resource, Studio automatically loads the editor associated with that resource into the Studio work space. Studio contains the following editors:

Editor	Use to...	See
Display Keys	Graphically create and define display keys.	“Using the Display Keys Editor” on page 137
Entity Names	Represent an entity name as a text string suitable for viewing in the PolicyCenter interface.	“Using the Entity Names Editor” on page 125
Gosu	Create and manage Gosu code used in classes, tests, enhancements, and interfaces.	“Working in the Gosu Editor” on page 108
Messaging	Work with messaging plugins.	“Using the Messaging Editor” on page 131
Page Configuration (PCF)	Graphically define and edit page configuration (PCF) files, used to render the PolicyCenter Web interface.	“Using the PCF Editor” on page 289
Plugins	Graphically define, edit and manage Java and Gosu plugins.	“Using the Plugins Registry Editor” on page 109

Editor	Use to...	See
Product Model	Define and configure the PolicyCenter product model.	"Using Product Designer to Edit the Product Model" on page 108
System Tables	Define PolicyCenter system tables.	"System Tables" on page 75 in the <i>Product Model Guide</i>
TypeList	Define TypeLists for use in the application.	"Working with TypeLists" on page 265
Workflows	Graphically define and edit application workflows.	"Using the Workflow Editor" on page 367

Working in the Gosu Editor

You use the Gosu editor to manage all code written in Gosu. If you open any of the following from the **Resources** pane, Studio automatically opens the file in the Gosu editor:

- Classes
- Enhancements
- Interfaces
- GUnit tests

See also

- "Classes" on page 191 in the *Gosu Reference Guide*

Using Product Designer to Edit the Product Model

Guidewire Product Designer provides a means of viewing, manipulating, and managing the PolicyCenter product model using a graphical interface (rather than through configuration files). By using Product Designer, you can define and configure the following:

- Products
- Policy Lines
- Question Sets
- Audit Schedules
- System Tables

In Product Designer, you can view, edit, and create new instances of each product model component through a web interface, in either a single-user or multi-user environment.

See also

- "Configuring the Product Model" on page 13 in the *Product Model Guide*
- "Product Model Overview" on page 475 in the *Application Guide*



chapter 6

Using the Plugins Registry Editor

A PolicyCenter plugin is a mini-program that you can invoke to perform some task or calculate a result.

This topic includes:

- “What Are Plugins?” on page 109
- “Working with Plugins” on page 110
- “Working with Plugin Versions” on page 112

What Are Plugins?

PolicyCenter *plugins* are mini-programs (Gosu or Java classes) that PolicyCenter invokes to perform an action or calculate a result.

- An example of a plugin that calculates a result is a policy number generation plugin, which PolicyCenter invokes to generate a new policy number as necessary.
- An example of a plugin that performs an action would be a message transport plugin, the purpose of which is to send a message to an external system.

Plugin Implementation Classes

A Guidewire plugin class implements a specific plugin interface. Guidewire provides a set of plugin interfaces in the base configuration. You can create a new class that implements the plugin interface for your business needs. You can choose to implement a plugin as either a Gosu class, Java class, or OSGi bundle. Alternatively, if the PolicyCenter base configuration already provides a implementation of the plugin interface, then you can register it.

See also

- “Plugin Overview” on page 123 in the *Integration Guide*.

What is the Plugins Registry?

Within Studio, expand the **configuration** → **config** → **Plugins** → **registry** node to view the contents of the Plugins Registry. Each item in the Plugins Registry is a .gwp file that represents a plugin implementation in the base configuration. To configure a particular plugin, double-click its name in the registry to enter the Plugins Registry editor.

Each Plugins Registry item (each .gwp file) includes fields for the following information:

- **Plugin name** – A unique name for this plugin implementation. If the plugin interface supports only a single implementation, make this the name of the interface without the package.
- **Implementation class** – The plugin implementation class as a fully-qualified class name.
- **Plugin interface** – The interface that the class implements. If the plugin interface field is left blank, PolicyCenter uses the plugin name as the interface name.

The Plugins Registry fields work slightly differently depending on whether the interface supports multiple implementations. Most plugin interfaces supports only a single plugin implementation. Other plugin interfaces, such as messaging plugins and startable plugins, support multiple plugin implementations.

See also

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

Startable Plugins

To register code that runs at server start up, you register startable plugin implementations. Startable plugins implement the `IStartablePlugin` interface. Typically, startable plugins are implemented as daemons, such as listeners to JMS queues. Unlike standard Guidewire plugins, you can stop and start startable plugins from the administrative interface. Alternatively, you can use PolicyCenter multi-threaded inbound integration APIs, which use startable plugins.

See also

- “Startable Plugins Overview” on page 259 in the *Integration Guide*
- “Multi-threaded Inbound Integration Overview” on page 267 in the *Integration Guide*

Working with Plugins

Creating a Plugins Registry Item

To create a plugin item

1. In the Project window, navigate to `configuration` → `config` → `Plugins` → `registry`.
2. Right-click `registry`, and then click `New` → `Plugin`.
3. In the `Name` text box, type the plugin name. If the interface supports only a single implementation, use the same name as the plugin interface. For the maximum supported implementations for each interface, see the table “Summary of All PolicyCenter Plugins” on page 141 in the *Integration Guide*.
4. In the `Interface` box, type the name of the plugin interface, or click `Browse`  to search for valid interfaces. For all startable plugins, enter `IStartablePlugin`.

Adding an Implementation to a Plugins Registry Item

To add a plugin implementation to a plugin item

1. In the Project window, navigate to `configuration` → `config` → `Plugins` → `registry`. In the list of plugin items in the Plugins Registry, double-click the plugin name to open it in the Plugins Registry editor.

2. Click **Add Plugin** , and then click the type of plugin to add: Gosu, Java, or OSGi.

Note: Do not change the value of the **Name** field in this editor. To rename the plugin implementation, right-click it in the Project window hierarchy, and then click **Refactor → Rename**.

Gosu Implementations

If you select **Add Gosu Plugin**, you see the following:

Gosu Class	Enter the name of the Gosu class that implements this plugin interface. In the base configuration, Guidewire places all Gosu plugin implementation classes in the following package in the Classes folder: <code>gw.plugin.package.impl</code> You must enter the fully-qualified path to the Gosu implementation class. For example, use <code>gw.plugin.email.impl.EmailMessageTransportPlugin</code> for the Gosu EmailMessageTransport plugin. See "Example Gosu Plugin" on page 129 in the <i>Integration Guide</i> for more information.
-------------------	--

Java Implementations

If you select **Add Java Plugin**, you see the following:

Java Class	Enter the fully qualified path to the Java class that implements this plugin. This is the dot separated package path to the class. Place all custom (non-Guidewire) Java plugin classes in the following directory: <code>PolicyCenter/modules/configuration/plugins/</code>
Plugin Directory	(Optional) Enter the name of the base plugin directory for the Java class. This is a folder (directory) in the <code>modules/configuration/plugins</code> directory. If you do not specify a value, Studio assumes that the class exists in the <code>modules/configuration/plugins/shared</code> directory. See "Special Notes For Java Plugins" on page 130 in the <i>Integration Guide</i> .

OSGi Implementations

If you select **Add OSGi Plugin**, you see the following:

Service PID	Enter the fully-qualified Java class name for your OSGi implementation class. See "Overview of Java and OSGi Support" on page 627 in the <i>Integration Guide</i> .
--------------------	--

After creating the plugin, you can add parameters to it. To do so, click **Add Parameter** , and then enter the parameter name and value.

If you have already set the environment or server property at the global level, then those values override any that you set in this location. For any property that you set in this location to have an effect, that property must be set to the **Default (null)** at the global level for this plugin. For more information on setting environment or server properties, see "Setting Environment and Server Context for Plugin Implementations" on page 112.

Enabling and Disabling a Plugin Implementation

You can choose to make a plugin implementation active or inactive using the **Enabled** checkbox. You can, for example, enable the plugin implementation for testing and disable it for production. It is important to understand, however, that you can still access a disabled plugin implementation and call it from code. Enabling or disabling a plugin implementation is only meaningful for plugins that care about the distinction. For example, you must enable a plugin for use in messaging in order for the plugin to work and for messages to reach their destination. If it is a concern, then the plugin user must determine whether a plugin is enabled.

If you change the status of the plugin (from enabled to disabled, or the reverse), then you must restart the application server for it to pick up this change.

Setting Environment and Server Context for Plugin Implementations

Within the Plugins Registry editor, you can set the plugin deployment environment (the **Environment** property) and the server ID (the **Server** property).

- Use **Environment** to set the deployment environment in which this plugin is active. For example, you may have multiple deployment environments (a test environment and a development environment) and you want this plugin to be active in only one of these environments.
- Use **Server** to set a specific server ID. For example, if running PolicyCenter in a clustered environment, you may want this plugin to be active only on a certain machine within the cluster.

These are *global* properties for this plugin. You can set either of these two properties on individual plugin properties. But, if set in this location, these override the individual settings.

See also

- “Reading System Properties in Plugins” on page 139 in the *Integration Guide*
- “Specifying Environment Properties in the <registry> Element” on page 15 in the *System Administration Guide*

Customizing Plugin Functionality

If you want to modify the behavior of a plugin, then do one of the following:

- Modify the underlying class that implements the plugin functionality.
- Change the plugin definition to point to an entirely different Java or Gosu plugin class.

For information on plugins in general, see “Plugin Overview” on page 123 in the *Integration Guide*.

For information on creating and deploying a specific plugin type, see the following topics:

Plugin type	Description	See
Gosu	A Gosu class	• “Example Gosu Plugin” on page 129 in the <i>Integration Guide</i>
Java	A Java class that does not use the OSGi standard.	• “Special Notes For Java Plugins” on page 130 in the <i>Integration Guide</i> • “Overview of Java and OSGi Support” on page 627 in the <i>Integration Guide</i>
OSGi	A Java class inside an OSGi bundle.	• “Overview of Java and OSGi Support” on page 627 in the <i>Integration Guide</i>

Working with Plugin Versions

If your installation includes more than one Guidewire application, be aware that some plugins exist primarily to connect to other Guidewire applications. If you want to use the base configuration plugin implementation to connect to other Guidewire applications, you must ensure that you use the correct version of the plugin implementation. The name of the classes are equivalent but vary in their package, which includes the application version number.

For the package name, Guidewire includes the application two-digit abbreviation followed by the application version number with no periods. For PolicyCenter 8.0.0, for example, the package includes pc800.

For example, in the Guidewire PolicyCenter application, the plugin implementation that connects PolicyCenter 8.0.0 to BillingCenter 8.0.0 is at the fully qualified path:

```
gw.plugin.billing.bc800.BCBillingSystemPlugin
```

For integrations with other Guidewire InsuranceSuite applications, choose the plugin implementation class that matches the version of your applications. Choose the implementation with the proper version number of the other application (not the current application) in its package name.

Guidewire uses the following abbreviation conventions for naming its applications:

Application	Abbreviation
ClaimCenter	cc
PolicyCenter	pc
ContactManager	ab
BillingCenter	bc

Working with Web Services

This topic discusses how you define and configure web services within Guidewire Studio.

This topic includes:

- “Web Services and Guidewire Studio” on page 115
- “Using the Web Service Editor” on page 116

Web Services and Guidewire Studio

This type of web service exists as a resource type called a web service collection. WS-I web service resources appear in the same resource hierarchy as Gosu classes. The location of the web service collection in the package hierarchy defines the package for the types that Gosu creates from the associated WSDL.

To create a WS-I web service in Studio by doing the following:

1. Navigate to a package under the `configuration → config → gsrc → wsi` folder.
2. Right-click the package and click `New → WebService Collection`.

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 in the *Integration Guide*.

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

Using the Web Service Editor

PolicyCenter provides a fully WS-I standard-compliant web services layer for both server (publishing) and client (consuming) web service APIs.

Studio manages WS-I web service resources as a resource type called a *web service collection*. The location of the web service collection in the package hierarchy defines the package for the types that Gosu creates from the associated WSDL. These WS-I web service resources appear in the same resource hierarchy as Gosu classes.

PolicyCenter supports both the SOAP 1.1 and SOAP 1.2 protocols. Guidewire recommends, however, the you use the SOAP 1.2 protocol as the preferred protocol.

The **WS-I Collections** editor consists of several different areas and items:

Area or item	Description
Resources	Displays the resource URLs that you have defined. Each URL points to a web service WSDL file. The WSDL file can exist in any of the following locations: <ul style="list-style-type: none"> • On the local file system • On a local server • On a remote server
Add Resource Remove Resource Fetch Updates	Located directly under the Resources pane, the function buttons consist of the following: <ul style="list-style-type: none"> • Add Resource – Use to enter the URL to the web service WSDL file. • Remove Resource – Use to remove a URL from the list of web service resources. • Fetch Updates – Use to retrieve XSD and WSDL files for a web service resource. You view these files in the Fetched Resources tab.
Settings	Use to add a setting for the following: <ul style="list-style-type: none"> • Override URL – Use to enter an override (proxy) URL for a defined web service resource. • Configuration Provide – Use to enter the name of a type that implements the <code>IWsWebServiceConfigurationProvider</code> interface. See “Adding Configuration Options” on page 82 in the <i>Integration Guide</i> for more information.
Fetched Resources	Shows the XSD and WSDL files pulled from the remote host that are associated with the listed resources.

See also

- “Calling a PolicyCenter Web Service from Java” on page 69 in the *Integration Guide*
- “Loading WSDL Directly into the File System” on page 76 in the *Integration Guide*
- “Adding Configuration Options” on page 82 in the *Integration Guide*
- “Defining a Web Service Collection” on page 116

Defining a Web Service Collection

To consume an external web service, you must load the associated WSDL and schema files for the web service into the local name space. You do this by defining a *web service collection* in PolicyCenter Studio. In the base configuration, Guidewire provides a number of default web service collections in the following location:

```
configuration → gsrc → wsi → local → gw → ...
configuration → gsrc → wsi → remote → gw → ...
```

Guidewire recommends that you define your web service collections within this directory structure as well.

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**.

To create a web service WSDL

1. Within Studio, navigate within the **wsi** hierarchy to a package in which to store your files.
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. 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 have modified the list of resource URLs and offers to fetch update 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 resource.

Implementing QuickJump Commands

This topic discusses how you can configure, or create new, QuickJump commands.

This topic includes:

- “What Is QuickJump?” on page 119
- “Adding a QuickJump Navigation Command” on page 120
- “Checking Permissions on QuickJump Navigation Commands” on page 122

What Is QuickJump?

The **QuickJump** box is a text-entry box for entering navigation commands using keyboard shortcuts. Guidewire places the box at the upper-right corner of each PolicyCenter screen. You set which commands are valid through the **QuickJump configuration** editor. At least one command must exist (be defined) in order for the **QuickJump** box to appear in PolicyCenter. (Therefore, to remove the **QuickJump** box from the PolicyCenter interface, remove all commands from the QuickJump configuration editor.)

You set the keyboard shortcut that activates the **QuickJump** box in `config.xml`. The default key is “/” (a forward slash). Therefore, the default action to access the box is `Alt+{/}`.

There are three basic types of navigation commands:

Type	Use for
QuickJumpCommandRef	Commands that navigate to a page that accepts one or more static (with respect to the command being defined) user-entered parameters. See “Implementing QuickJumpCommandRef Commands” on page 120 for details.
StaticNavigationCommandRef	Commands that navigate to a page without accepting user-entered parameters. See “Implementing StaticNavigationCommandRef Commands” on page 122.
ContextualNavigationCommandRef	Commands that navigate to a page that takes a single parameter, with the parameter determined based on the user’s current location. See “Implementing ContextualNavigationCommandRef Commands” on page 122.

Adding a QuickJump Navigation Command

If you add a command, first set the command type, then define the command by setting certain parameters. The editor contains a table with each row defining a single command and each column representing a specific command parameter. You use certain columns with specific command types only. PolicyCenter enables only those row cells that are appropriate for the command, meaning that you can only enter text in those specific fields.

Column	Only use with	Description
Command Name	<ul style="list-style-type: none"> • QuickJumpCommandRef • StaticNavigationCommandRef • ContextualNavigationCommandRef 	Display key specifying the command string the user types to invoke the command. Note that the command string must not contain a space.
Command Class	<ul style="list-style-type: none"> • QuickJumpCommandRef 	Class that specifies how to implement the command. This class must be a subclass of QuickJumpCommand. Guidewire intentionally makes the base QuickJumpCommand class package local. To implement, override one of the subclasses described in Implementing QuickJumpCommandRef Commands.
		You only need to subclass QuickJumpCommand if you plan to implement the QuickJumpCommandRef command type. For the other two command types, you use the existing base class appropriate for the command—either StaticNavigationCommand or ContextualNavigationCommand—and enter the other required information in the appropriate columns.
Command Target	<ul style="list-style-type: none"> • StaticNavigationCommandRef • ContextualNavigationCommandRef 	Target page ID.
Command Arguments	<ul style="list-style-type: none"> • StaticNavigationCommandRef 	Comma-separated list of parameters used in the case in which the target page accepts one or more string parameters. (This is not common.)
Context Symbol	<ul style="list-style-type: none"> • ContextualNavigationCommandRef 	Name of a variable on the user's current page.
Context Type	<ul style="list-style-type: none"> • ContextualNavigationCommandRef 	Type of context symbol (variable).

Implementing QuickJumpCommandRef Commands

To implement the QuickJumpCommandRef navigation command type, subclass QuickJumpCommand or one of its existing subclasses. See the following sections for details:

Subclass	Section
StaticNavigationCommand	Navigation Commands with One or More Static Parameters
ParameterizedNavigationCommand	Navigation Commands with an Explicit Parameter (Including Search)
ContextualNavigationCommand	Navigation Commands with an Inferred Parameter
EntityViewCommand	Navigation to an Entity-Viewing Page

All QuickJumpCommand subclasses must define a constructor that takes a single parameter—the command name—as a String.

Navigation Commands with One or More Static Parameters

To perform simple navigation to a page that accepts one or more parameters (which are always the same for a given command), subclass StaticNavigationCommand. The class constructor must call the super constructor, which takes the following arguments:

- The command name (which you pass into your subclass's constructor)
- The target location's ID

Your subclass implementation must override the `getLocationArgumentTypes` and `getLocationArguments` methods to provide the required parameters for the target location.

It is possible to create a no-parameter implementation by subclassing `StaticNavigationCommand`. However, Guidewire recommends that you use the `StaticNavigationCommandRef` command type instead as it reduces the number of extraneous classes needed. See “[Implementing StaticNavigationCommandRef Commands](#)” on page 122 for details.

Navigation Commands with an Explicit Parameter (Including Search)

To create a command that performs simple navigation to a page that accepts a single user parameter, subclass `ParameterizedNavigationCommand`. The constructor takes the same two arguments as `StaticNavigationCommand`. Your subclass must override the `getParameterSuggestions` method, which provides the list of auto-complete suggestions for the parameter. It must also override the `getParameter` method, which creates or fetches the actual parameter object given the user's final input.

Subclasses of `ParameterizedNavigationCommand` must also implement `getCommandDisplaySuffix`.

PolicyCenter displays the command in the `QuickJump` box as part of the auto-complete list (before the user has entered the entire command). Therefore, PolicyCenter displays the command name followed by the command display suffix. This is typically some indication of what the parameter is, for example *bean name* or *policy number*.

Navigation Commands with an Inferred Parameter

To implement a command that navigates to a page that accepts a single parameter, with the parameter based on the user's current location, subclass `ContextualNavigationCommand`. The constructor takes the same two arguments as `StaticNavigationCommand`, plus two additional arguments:

- The name of a PCF variable. If this variable exists on the user's current location, Studio makes the command available and uses the value of the variable as the parameter to the target location.
- The type of the variable.

Guidewire recommends, however, that you use the `ContextualNavigationCommandRef` command type instead of subclassing `ContextualNavigationCommand`. See “[Implementing ContextualNavigationCommandRef Commands](#)” on page 122 for details.

Navigation to an Entity-Viewing Page

For commands that navigate to a page that simply displays information about some entity, subclass `EntityViewCommand`. The constructor takes the following arguments:

- The name of the command (which you pass into your subclass's constructor)
- The type of the entity
- A property on the entity to use in matching the user's input (and providing auto-complete suggestions)
- The permission key that determines whether the user has permission to know the entity exists (This is typically a “view” permission.)
- The target location's ID

Subclasses must override `handleEntityNotFound` to specify behavior on incomplete or incorrect user input. A typical implementation simply throws a `UserDisplayableException`. Subclasses must also implement `getCommandDisplaySuffix`, which behaves in the fashion described previously in “[Navigation Commands with an Explicit Parameter \(Including Search\)](#)” on page 121.

By default, parameter suggestions and parameter matching use a query that finds all entities of the appropriate type in which the specified property starts with the user's input. If this query is too inefficient, the subclass can override `getQueryProcessor` (for auto-complete) and `findEntity` (for parameter matching). If you do not want some users to see the command, then the subclass must also override the `isPermitted` method.

By default, the auto-complete list displays each suggested parameter completion as the name of the command followed by the value of the matched parameter. Subclasses can override `getFullDisplay` to change this behavior. However, the suggested name must not stray too far from the default, as it does not change what appears in the **QuickJump** box after a user selects the suggestion. Entity view commands automatically chain to any appropriate contextual navigation command (for example, “Claim <claim #> Financials”).

Implementing StaticNavigationCommandRef Commands

`StaticNavigationCommandRef` specifies a command that navigates to a page without accepting user-entered parameters. It is the simplest to implement. You specify the Command Name and Command Target in exactly the same manner as for a static navigation command. You must also specify the Command Target, and any necessary Command Arguments. These parameters have the following meanings:

- Command Target specifies the ID of the target page.
- Command Arguments specify one or more parameters to use in the case in which the target page accepts one or more string parameters. If there is more than one parameter, enter a comma-separated list.

Implementing ContextualNavigationCommandRef Commands

`ContextualNavigationCommandRef` specifies a command that navigates to a page that takes a single parameter. (The user's current location determines the parameter.) You specify the Command Name and Command Target in exactly the same manner as for a static navigation command. You must also specify the Context Symbol and the Context Type. These parameters have the following meanings:

- Context Symbol specifies that name of a variable on the user's current page.
- Context Type specifies that variable's type.

PolicyCenter passes the value of this variable to the target location as the only parameter. If no such variable exists on the current page, then the command is not available to the user and the command does not appear in the **QuickJump** box's auto-complete suggestions.

If the Context Type is an entity, then any EntityViewCommands matching the entity type can automatically be “chained” by the user into the `ContextualNavigationCommand`. (See “Navigation to an Entity-Viewing Page” on page 121 for more information.) For instance, suppose that there is an EntityViewCommand called `Policy` that takes a policy number and navigates to a `Policy`. Also, suppose that there is a `ContextualNavigationCommand` called `Contacts` whose context type is `Policy`. In this case, the user can type `Policy 35-402398 Contacts` to invoke the `Contacts` command on the specified `Policy`.

Checking Permissions on QuickJump Navigation Commands

Keep the following security issues in mind as you create navigation commands for the **QuickJump** box.

Subclassing StaticNavigationCommand

Commands that implement this subclass check the `canVisit` permission by default to determine whether a user has the necessary permission to see that QuickJump option in the **QuickJump** box. The permission hole in this case arises if permissions were in place for all approaches to the destination but not on the destination itself.

For example, suppose that you create a new QuickJump navigation for `NewNotePopup`. Then suppose that previously you had placed a permission check on all `New Note` buttons. In that case PolicyCenter would have checked the `Note.create` permissions. However, enabling QuickJump navigation to `NewNotePopup` bypasses those previous permissions checks. The best practice is to check permissions on the `canVisit` tag of the actual destination page, in this case, on `NewNotePopup`.

Subclassing ContextualNavigationCommand

As with `StaticNavigationCommand` subclasses, add permission checks to the destination page's `canVisit` tag.

Subclassing ParameterizedNavigationCommand

Classes subclassing `ParameterizedNavigationCommand` have the (previously described) method called `isPermitted`, which is possible for you to override. This method—`isPermitted`—controls whether the user can see the navigation command in the `QuickJump` box. After a user invokes a command, PolicyCenter performs standard permission checks (for example, checking the `canVisit` expression on the target page), and presents an error message to unauthorized users.

It is possible for the `canVisit` expression on the destination page to return a different value depending on the actual parameters passed into it. As a consequence, PolicyCenter cannot determine automatically whether to display the command to the user in the `QuickJump` box before the user enters a value for the parameter. If it is possible to manually determine whether to display the command to the user, check for permission using the overridden `isPermitted` method. (This might be, for example, from the destination's `canVisit` attribute.)

Using the Entity Names Editor

This topic describes entity names and entity name types, and how to work with the entity names in the Studio **Entity Names** editor.

This topic includes:

- “Entity Names Editor” on page 125
- “Variable Table” on page 126
- “Gosu Text Editor” on page 128
- “Including Data from Subentities” on page 128
- “Entity Name Types” on page 129

Entity Names Editor

It is possible to define an entity name as text string, which you can then use in the PolicyCenter interface to represent that entity. Thus, you often see the term *display name* associated with this feature as well, especially in code and in GosuDoc.

PolicyCenter uses the `DisplayName` property on an entity to represent the entity name as a text string. You can define this entity name string as a simple text string or use a complicated Gosu expression to generate the name. PolicyCenter uses these entity name definitions in generating database queries that return the limited information needed to construct the display name string. This ensures that PolicyCenter does not load the entire entity and its subentities into memory simply to retrieve the few field values necessary to generate the display name.

The use of the *Entity Name* feature helps to avoid loading entities into memory unless you are actually going to view or edit its details. The use of display names improves overall application performance.

The **Entity Names** editor consists of two parts:

- A table in which you manage variables for use in the Gosu code that defines the entity name
- A Gosu editor that contains the Gosu code that defines the entity name

To deploy your changes, you must stop and restart the application server.

Variable Table

You must declare any field that you reference in the entity definition (in the code definition pane) as a variable in the variable table at the top of the page. This tells the Entity Name feature which fields to load from the database, and puts each value in a variable for you to use.

For example, the Contact entity name defines the following variables:

Name	Entity Path	Sort Path	Sort Order	Use Entity Name?
SubType	Contact.SubType	Contact.SubType		
LastName	Person.LastName	Person.LastNameDenorm	1	
FirstName	Person.FirstName	Person.FirstNameDenorm	2	
Suffix	Person.Suffix		3	
Name	Company.Name	Company.NameDenorm	4	

Notice that this defines LastName as Person.LastName and Name as Company.Name, for example.

Use the variable table to manage variables that you can embed in the Gosu entity name definitions. You can add, duplicate, and remove variables using the function buttons by the table. The columns in the table have the following meanings:

Name	Name of the variable
Entity Path	Entity.property that the variable represents
Sort Path	Defines the values that PolicyCenter uses in a sort
Sort Order	Defines the order in which PolicyCenter sorts the Sort Path values
Use Entity Name?	Sets whether to use this value as the entity display name

The Entity Path Column

Use only actual columns in the database as members of the **Entity Path** value. You must declare an actual database column in metadata, in an actual definition file. If you do not define a column in metadata, then PolicyCenter labels that entity column (field) as virtual in the *PolicyCenter Data Dictionary*.

Thus:

- You cannot use ClaimContactRole fields in the **Entity Path**, such as Exposure.Incident.Injured. This is because the Injured contact role on Incident does not have a denormalized column.
- You can, however, use special denormalized fields for certain claim contacts, such as Exposure.ClaimantDenorm or Claim.InsuredDenorm. The description of the column indicate which ClaimContactRole value it denormalizes.

The Use Entity Names? Column

The last column in the variable table is **Use Entity Name?** The column takes a Boolean **true/false** value, or the column can be empty.

- A value of **true** is meaningful only if the value of **Entity Path** is an entity type. A value of **true** instructs the Entity Name utility to calculate the Entity Name for that entity, instead of loading the entity into memory. The variable for that subentity is of type **String** and you can use the variable in the Gosu code that constructs the current Entity Name.

Note: If the value of **Entity Path** is an entity, then you must set the value of **Use Entity Type?** to **true**. Otherwise, a variable entry that ends in an entity value uploads that entire entity, which defeats the purpose of using Entity Names.

- A value of **false** indicates that PolicyCenter does not use the **Entity Path** value as an entity display name.
- An empty column is the same as a value of **false**. This is the default.

Set the **Use Entity Name?** value to **true** if you want to include the entire Entity Name for a particular subentity. For example, suppose that you are editing the Exposure entity name and that you create a variable called **claimant** with an **Entity Path of Exposure.ClaimantDenorm**. Suppose also that you set the value of **Use Entity Name** to **true**. In this case, the entity name for the Claimant, as defined by the Contact entity name definition, would be included in a **String** variable called **claimant**. PolicyCenter would then use this value in constructing the entity name for the **Exposure** entity.

Note: If you set the **Use Entity Name?** field to **true** and then attempt to use a virtual field as an **Entity Path** value, Studio resource verification generates an error.

Evaluating Null Values

If the value of **Use Entity Name** is **true**, then PolicyCenter always evaluates the entity name definition, even if the foreign key is **null**. By convention, in this case, the entity name definition usually returns the empty string **" "**. In other words, the entity name string can never be **null** even if the foreign key is **null**. You can use the **HasContent** enhancement property on **String** to test whether the display name string is empty.

Thus, as you write entity name definitions, Guidewire recommends that you return the empty string if all the variables in your entity name definition are **null** or empty. Guidewire uses the empty string (instead of returning **null**) to prevent Null Pointer Exceptions. For example, suppose that you construct an entity name such as "X-Y-Z", in which you add a hyphen between variables X,Y, and Z from the database. In this case, be sure you return the empty string **" "** if X,Y, and Z are all **null** or empty and not **" - - "**.

The Sort Columns

The two columns **Sort Path** and **Sort Order** do not, strictly speaking, involve variable replacement in the entity name Gosu code. Rather, you use them to define how to sort beans of the same entity.

Sort Path	Defines the values that PolicyCenter uses in a sort
Sort Order	Defines the order in which PolicyCenter sorts the Sort Path values

Therefore, if PolicyCenter is in the process of determining how to order two contacts, it first compares the values in the (**Sort Path**) **LastNamesDenorm** fields (**Sort Order = 1**). If these values are equal, Studio then compares the values in the **FirstNamesDenorm** fields (**Sort Order = 2**), repeating this process for as long as there are fields to compare.

These columns specify the default sort order. Other aspects of Guidewire PolicyCenter can override this sort order, for example, the sort order property of a list view cell widget.

Gosu Text Editor

You enter the actual Gosu code used to construct the entity name in the code definition pane underneath the variable table. Studio then replaces the variable with mapped property.

The following Gosu definition code for the Contact entity name shows these mappings.

```
var retString = ""

if ( SubType != null && Person.isAssignableFrom( Type.forName("entity." + SubType) ) ) {
    if (FirstName != null and FirstName.length() > 0) {
        retString = retString + FirstName + " "
    }
    if (LastName != null and LastName.length() > 0) {
        retString = retString + LastName + " "
    }
    if (Suffix != null) {
        retString = retString + gw.api.util.TypeKeyUtil.toDisplayName(Suffix) + " "
    }
} else {
    retString = Name != null and Name.length() > 0 ? Name : ""
}
return retString
```

To use the Contact entity name definition, you can embed the following in a PCF page, for example.

```
<Cell id="Name" value="contact.DisplayName" ... />
```

Including Data from Subentities

Many times, you want to include information from subentities of the current entity in its Entity Name. For example, this happens often with Contacts related to the current entity. Guidewire recommends that you do one of the following to include data from a subentity. (The two options are mutually exclusive. You must do one or the other.)

Option 1: Use the DisplayName for a Subentity

To use the `DisplayName` value for a subentity, you must set the value of `Use Entity Name` to `true` on the variable definition. For example, for Contacts, you must set the value to `true` through an explicit `Denorm` column, such as `Exposure.ClaimantDenorm`.

To illustrate:

Name	Entity Path	Use Entity Name?
claimantDisplayName	Exposure.ClaimantDenorm	true
incidentDisplayName	Exposure.Incident	true

Option 2: Reference Fields on the Subentity

It is possible that you do not want to use the Entity Name as defined for the subentity's type. If so, then you need to set up variables in the table to obtain the fields from the subentity that you need. To illustrate:

Name	Entity Path	Use Entity Name?
claimantFirstName	Exposure.ClaimantDenorm.FirstName	false
claimantLastName	Exposure.ClaimantDenorm.LastName	false
severity	Exposure.Incident.Severity	false
incidentDesc	Exposure.Incident.Description	false

You can then use these variables in Gosu code (in the text editor) to include the Claimant and Incident information in the entity name for Exposure.

Guidewire Recommendations

Do not end an **Entity Path** value with an entity foreign key, without setting the **Use Entity Name** value to **true**. Otherwise, PolicyCenter loads the entire entity being referenced into memory. In actuality, you probably only need a couple fields from the entity to construct your entity name. Instead, you one of the approaches described in one of the previous steps.

Denormalized Columns

Within the PolicyCenter data model, it is possible for a column to end in **Denorm** for (at least) two different reasons:

- The column contains a direct foreign key to a particular **Contact** (for example, as in `Claim.InsuredDenorm`.)
- The original column is of type **String** and the column attribute `supportsLinguisticSearch` is set to **true**. In this case, the denormalized column contains a normalized version of the string for searching, sorting, and indexing. Thus, the **Contact** entity definition uses `LastNameDenorm` and `FirstNameDenorm` as the sort columns in the definition for the **Contact** entity name. It then uses `LastName` and `FirstName` in the variables' entity paths for eventual inclusion in the entity name string.

Entity Name Types

Guidewire calls an entity name definition the entity name *type*. Thus, most—but not all—entity names have a single type. However, it is possible for certain entities in the base application to have multiple, alternate names (types) and thus, multiple entity name definitions. Again, these can be either simple text strings, or more complicated Gosu expressions.

Studio displays each entity name type as a separate tab or code definition area at the bottom of the screen. You cannot add or delete an entity name type to the base application. You can, however, change the Gosu definition of an entity name type. Guidewire recommends, however, that you not modify an entity name type definition without a great deal of thought.

Most entity names have only the single type named *Default*. You can access the *Default* entity name from Gosu code by using the following code:

```
entity.DisplayName
```

Only internal application code (internal code that Guidewire uses to build the application) can access any of non-default entity name types. For example, some of the entity names contain an additional type or definition of `ContactRoleMessage`. PolicyCenter uses the `ContactRoleMessage` type to define the format of the entity name to use in role validation error messages. In some cases, this definition is merely the same as the default definition.

Note: It is not possible for you to either add or delete an entity name type from the base application configuration. You can, however, modify the definition—the Gosu code—for all defined types. You can directly access only the default type from Gosu code.

Using the Messaging Editor

This topic covers how you use the **Messaging** editor in Guidewire Studio.

This topic includes:

- “**Messaging Editor**” on page 131

Messaging Editor

You use the **Messaging** editor to set up and define one or more message environments, each of which includes one or more message destinations. A message destination is an abstraction that represents an external system. Typically, a destination represents a distinct remote system. However, you can also use destinations to represent different remote APIs or different types of messages that must be sent from PolicyCenter business rules.

You use the **Messaging** editor to set up and define message destinations, including the destination ID, name, and the transport plugin to use with this destination. In a similar fashion to the **Plugins** editor, you can also set the deployment environment in which this message destination is active.

Each destination can specify a list of events that are of interest to it, along with some basic configuration information.

See also

- “Message Destination Overview” on page 302 in the *Integration Guide*
- “Implementing Messaging Plugins” on page 333 in the *Integration Guide*
- “Messaging and Events” on page 289 in the *Integration Guide*

Adding a Messaging Environment

You can define multiple messaging environments to suit different purposes. For example, you can set up different messaging environments for the following:

- A development environment
- A test environment

- A production environment

Guidewire provides a single default messaging environment in the PolicyCenter base configuration. You see it listed as **Default** in the **Messaging Config** drop-down list.

To create a new messaging environment

1. Next to the **Messaging Config** drop-down list, click **Add Messaging** .
2. In the **New Messaging** dialog box, type the name for the new message environment.
3. Add message destinations as required. See “Adding a Message Destination” on page 132 for details.

To remove a messaging environment

1. Next to the **Messaging Config** drop-down list, click the messaging environment to remove.
2. Click **Remove Messaging** .

PolicyCenter disables the messaging environment **Remove** button if only one message environment exists. However, if there are several messaging environments, and you have added message destinations to each environment, then there are multiple **Remove** options available. The **Remove Messaging** button at the top of the screen removes the currently selected message environment. The other **Remove Destination** option removes the currently selected message destination from the list of destinations.

Note: Be careful not to inadvertently click the top **Remove Messaging** button, as PolicyCenter deletes the message environment without any additional warning. You cannot undo this action.

Adding a Message Destination

To add a message destination, open the **Messaging** editor, select a message environment, click **Add Destination**, and fill in the required fields. Notice that Studio requires that you enter a plugin name, for example, for the Transport plugin.

It is important to understand the difference between the implementation class name and the plugin name. After you write code that implements a messaging plugin, you must register it in Studio. As you register an implementation of the 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. (For example, it is possible to create multiple distinct messaging and encryption plugins.)

After you click **Add Destination** in the **Messaging** editor, fill in the following fields.

ID	The destination ID (as an integer value). The valid range for custom destination IDs is 0 through 63, inclusive. Guidewire reserves all other destination IDs for built-in destinations such as the email transport destination. Studio marks these internal values with a gray background, indicating that they are not editable. Studio also marks valid entries with a white background and invalid entries with a red background. For more information on message IDs, see: <ul style="list-style-type: none">“Implementing a Message Transport Plugin” on page 334 in the <i>Integration Guide</i>
Name	The name to use for this messaging destination.
Transport Plugin	The name of the <code>MessageTransport</code> plugin implementation that knows how to send messages for this messaging destination. A destination must define a message transport plugin that sends a <code>Message</code> object over a physical or abstract transport. For example, the plugin might do one of the following: <ul style="list-style-type: none">Submit the message to a message queueCall a remote web service API and get an immediate response that the system handled the messageImplement a proprietary protocol that is specific to a remote system For more information, see the following: <ul style="list-style-type: none">“Messaging Overview” on page 290 in the <i>Integration Guide</i>“Implementing a Message Transport Plugin” on page 334 in the <i>Integration Guide</i>

If you select a specific row in the message ID table, you see additional fields. These fields have the following meanings:

Field	Description
Request Plugin	<p>A destination can optionally define a message request (<code>MessageRequest</code>) plugin to prepare or pre-process a <code>Message</code> object before a message is sent to the message transport. For example, the <code>MessageRequest</code> plugin can:</p> <ul style="list-style-type: none"> Translate strings or codes in a text-type message payload to codes for a remote system. Translate name/value pairs in a text-type message payload into XML. Set messaging-specific data model extension properties on the <code>Message</code> object before sending it. <p>To use a message reply plugin, in this Messaging editor field, type the name of the <code>MessageRequest</code> plugin implementation. If the destination requires no special message preparation, omit the request plugin entirely for the destination.</p> <p>For implementation details, see the following:</p> <ul style="list-style-type: none"> "Implementing a Message Request Plugin" on page 334 in the <i>Integration Guide</i> "Message Destination Overview" on page 302 in the <i>Integration Guide</i>
Reply Plugin	<p>A destination can optionally define a message reply (<code>MessageReply</code>) plugin to asynchronously acknowledge a <code>Message</code> object. For instance, this plugin can implement a trigger from an external system to notify PolicyCenter that the message send succeeded or failed. To use a message reply plugin, in this Messaging editor field, type the name of the <code>MessageReply</code> plugin implementation. If the destination requires no asynchronous acknowledgement or asynchronous post-processing, omit the reply plugin configuration settings.</p> <p>For implementation details, see the following:</p> <ul style="list-style-type: none"> "Implementing a Message Reply Plugin" on page 336 in the <i>Integration Guide</i> "Message Destination Overview" on page 302 in the <i>Integration Guide</i>
Alternative Primary Entity	<p>In PolicyCenter, for each message destination, messages associated with accounts are always sent ordered by account. This is known as <i>safe-ordered</i> messages.</p> <p>Use this field to provide a second or alternative entity on which to safe-order messages. For example, you integrate Guidewire PolicyCenter with Guidewire ContactManager. It is possible that you want to safe-order messages by contact as well as by account. In so, add Contact to this field.</p> <p>See "Message Ordering and Multi-Threaded Sending" on page 323 in the <i>Integration Guide</i>.</p>
Chunk Size	<p>The number of messages that the messaging subsystem retrieves from the database in each round of sending, if possible. By default, Guidewire sets this value to 100,000. This number is usually sufficient to include all sendable messages currently in the send queue. Be careful not to set this number too low.</p> <p>For more information, see the following:</p> <ul style="list-style-type: none"> "Message Ordering and Multi-Threaded Sending" on page 323 in the <i>Integration Guide</i>
Poll Interval	<p>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, than the thread does not sleep at all and continues to the next round of querying and sending.</p> <p>For details on how the polling interval works, see the following:</p> <ul style="list-style-type: none"> "Message Ordering and Multi-Threaded Sending" on page 323 in the <i>Integration Guide</i> <p>NOTE The value you choose for the poll interval value can significantly affect messaging performance. If you change this value, carefully test the performance implications under realistic conditions. If your performance issues relate primarily to many messages for each account for each destination, then the polling interval is the most important messaging performance setting.</p>
Max Retries	The number of retries to attempt before the retryable error becomes non-retryable.
Initial Retry Interval	The amount of time (in milliseconds) to wait before attempting to retry sending a message after a retryable error condition occurs.

Field	Description
Number Sender Threads	<p>To send messages associated with an account (<i>safe-ordered</i> messages), 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. PolicyCenter ignores this setting for non-claim-specific messages, since those are always handled by one thread for each destination.</p> <p>If your performance issues primarily relate to many messages but few messages for each account for each destination, then this is the most important messaging performance setting.</p> <p>For more information, see the following:</p> <ul style="list-style-type: none"> “Message Ordering and Multi-Threaded Sending” on page 323 in the <i>Integration Guide</i>
Shutdown Timeout	<p>Messaging plugins have methods to handle the administrative commands suspend, resume, and preparing for the messaging system to shutdown. During the suspend, shutdown, and resume methods of the plugin, the plugin must not call any APIs that suspend or resume messaging destinations. (This includes—but is not limited to—IMessageToolsAPI web service APIs.) Doing so creates circular application logic. Guidewire disallows such actions.</p> <p>The shutdown timeout value is the length of time to wait before attempting to shutdown the messaging subsystem.</p> <p>For more information, see the following:</p> <ul style="list-style-type: none"> “Handling Messaging Destination Suspend, Resume, Shutdown” on page 339 in the <i>Integration Guide</i>.
Retry Backoff Multiplier	The amount to increase the time between retries, specified as a multiplier of the time previously attempted. For example, if the last retry time attempted was 5 minutes and you set this value to 2, PolicyCenter attempts the next retry in 10 minutes.

The **Messaging** editor contains several additional checkboxes directly underneath the message definition fields:

Checkbox	Description
Enabled	Select this checkbox to enable this message destination.
Strict mode	<p>Select this checkbox to ensure that:</p> <ul style="list-style-type: none"> PolicyCenter waits for acknowledgements for each non-safe-ordered message before sending the next one. The message sending system blocks all future messages of all types (both safe-ordered and non-safe-ordered) if there are errors in non-safe-ordered messages. <p>See “Message Ordering and Multi-Threaded Sending” on page 323 in the <i>Integration Guide</i> for more details.</p>

Associating Event Names with a Message Destination

To define one or more specific events for which you want this message destination to listen, click **Add Event**  under **Events**. Each event triggers the Event Fired rule set for that destination. Use the special event name wild-card string “`(\w)*`” to listen for all events.

To get notifications using Event Fired rules when specific types of data changes occur, you must specify one or more messaging destinations to listen for that event. If no messaging destination listens for an event, PolicyCenter does not call the Event Fired rules for that combination of event and destination.

If more than one destination listens for that event, the Event Fired rules run multiple times, varying only in the destination ID. To get the destination ID in your Event Fired rules, check the property `messageContext.destID`.

For much more information about events and the messaging system, refer to “Messaging and Events” on page 289 in the *Integration Guide*.

See also

- For a list of built-in events that PolicyCenter triggers, see “List of Messaging Events in PolicyCenter” on page 309 in the *Integration Guide*.

Using the Display Keys Editor

This topic discusses how to work with the display key editor that is available to you in Guidewire Studio.

This topic includes:

- “Display Keys Editor” on page 137
- “Creating Display Keys in a Gosu Editor” on page 138
- “Retrieving the Value of a Display Key” on page 138

Display Keys Editor

A `DisplayKey` represents a single user-viewable text string. Guidewire strongly recommends that any string literal that can potentially reach the eyes of the user be kept as a `DisplayKey` rather than a hard-coded `String` literal.

PolicyCenter stores each display key in a `display.properties` file. If there is no international localization, PolicyCenter stores this file in the following location:

`PolicyCenter/modules/configuration/config/locale/en_US`

However, if you do localize one or more display keys, then PolicyCenter uses additional `display.properties` files, one for each locale that you create. For more information, see “Localizing Display Keys” on page 45 in the *Globalization Guide*.

PolicyCenter represents display keys within a hierarchical name space. Within `display.properties`, this translates into a dot (.) separating the levels of the hierarchy.

Within the `Display Keys` editor, Studio does the following automatically:

- It sorts display keys alphabetically—both at the root level and at the package level—as you create the display key.
- It removes empty display keys—those for which no value was set—upon a save operation.

To access the `Display Keys` editor in Studio, in the Project window, navigate to `configuration → config → Localizations → en_US`, and then open the file `display.properties`.

You can also place your cursor in a text string and press Alt+Enter to open the **Create Display Key** dialog.

Using the **Display Keys** editor, you can do the following:

Task	Actions
View a display key	Navigate to the display key that you want to view by scrolling through the <code>display.properties</code> file. To search for a particular key or value, press Ctrl+F and then type your search term in the search bar.
Modify the text of an existing display key	Navigate to the display key that you want to modify, and then modify the string in the editor as you want.
Create a new display key	In the Display Key editor, type the desired name and value for your new display key.
Delete an existing display key	Highlight the display key that you want to delete, and then press Delete.
Localize an existing display key	Select a different locale and enter the localized text. See “ Localizing Display Keys ” on page 45 in the <i>Globalization Guide</i> .

Creating Display Keys in a Gosu Editor

You can also immediately create a display key from within a Gosu editor by entering a string literal. If you place the cursor within the string and then press Alt+Enter, Studio prompts you to create a new display key for that string.

For example, suppose that you enter the following in the **Rule Actions** pane in the Rules editor:

```
var errorString = "SendFailed"
```

If you place the cursor within that string, press Alt+Enter, and then click **Convert string literal to display key**, Studio opens the **Create Display Key** dialog. It also populates the **Display Key Name** field with the string text that you entered. The dialog contains a text entry field in which you can enter the localized text for the string that you entered in the Rules editor.

After you enter the text and click **OK**, Studio replaces the string literal with the new display key. For example:

```
var errorString = displaykey.SendFailed
```

Retrieving the Value of a Display Key

Some display keys contain a place holder argument or parameter, designated by `{0}`. PolicyCenter replaces each of these parameters with actual values at run time. For example, in the `display.properties` file, you see the following:

```
Java.Activities.Error.CannotPerformAction = You do not have permission to perform actions on the
following activities\: {0}.
```

Thus, at run time, PolicyCenter replaces `{0}` with the appropriate value, in this case, the name of an activity.

Occasionally, there are display keys that contain multiple arguments. For example:

```
Java.Admin.User.InvalidGroupAdd = The group {0} cannot be added for the user {1}
as they do not belong to the same organization.
```

Class `displaykey`

Use the `displaykey` class to return the value of the display key. Use the following syntax:

```
displaykey.[path to display key].[display key name]
```

For example:

```
displaykey.Java.Admin.User.DuplicateRoleError
```

returns

```
User has duplicate roles
```

This also works with display keys that require a parameter or parameters. To retrieve the parameter value, use the following syntax.

```
displaykey.[path to display key].[display key name](arg1)
```

For example, file `display.properties` defines the following display key with placeholder `{0}`:

```
Java.UserDetail.Delete.IsSupervisorError = Cannot delete user because that user is the supervisor  
of the following groups\: {0}
```

Suppose that you have the following display key code:

```
displaykey.Java.UserDetail.Delete.IsSupervisorError( GroupName )
```

If you have already retrieved a value for `GroupName`, this display key returns the following:

```
Cannot delete user because they are supervisor of the following groups: WesternRegion
```

The same syntax works with multiple arguments as well:

```
displaykey.[path to display key].[display key name](arg1, arg2, ...)
```


Data Model Configuration

Working with the Data Dictionary

Guidewire provides the *Data Dictionary* to help you understand the PolicyCenter data model. The *Data Dictionary* is a detailed set of linked documentation in HTML format. These linked HTML pages contain information on all the data entities and typelists that make up the current data model. The *Data Dictionary* also includes information on associated fields and their attributes for the data entities and data extension entities.

This topic includes:

- “What is the Data Dictionary?” on page 143
- “What Can You View in the Data Dictionary?” on page 144
- “Using the Data Dictionary” on page 144

What is the Data Dictionary?

The *Data Dictionary* documents all the entities and typelists in your PolicyCenter installation. Provided that you regenerate it following any customizations to the data model, the dictionary documents both the base PolicyCenter data model and your extensions to it. Using the *Data Dictionary*, you can view information about each entity, such as fields and attributes on it.

You must manually generate the Data Dictionary after you install Guidewire PolicyCenter. Guidewire strongly recommends that you perform this task as part of the installation process. Also, as you extend the data model, it is important that you regenerate the *Data Dictionary* as needed in order to view your extensions to the data model.

To generate the *PolicyCenter Data Dictionary*, run the following command from the `PolicyCenter/bin` directory:

```
gwpc regen-dictionary
```

PolicyCenter stores the current version of the *Data Dictionary* in the following directory:

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

To view the *Data Dictionary*, open the following file:

```
PolicyCenter/build/dictionary/data/index.html
```

See also

- “Regenerating the Data Dictionary and Security Dictionary” on page 32

What Can You View in the Data Dictionary?

Note: If you use a third-party tool to edit PolicyCenter configuration files, Guidewire recommends that you work with one that fully supports UTF-8 file encoding. If the editing tool does not handle UTF-8 characters correctly, it can create errors that you then see in the Guidewire *Data Dictionary*. This is not an issue with the *Data Dictionary*. It occurs only if the third-party tool cannot handle UTF-8 values correctly.

After you open the *Data Dictionary* (at `PolicyCenter/build/dictionary/data/index.html`), Guidewire presents you with multiple choices. For example, you can choose to view either **Data Entities** or **Data Entities (Migration View)**.

The standard and migration views are similar but not identical. You use each for a different purpose. In general:

- Use the *standard view* to view a full set of entities associated with the PolicyCenter application and the columns, typekeys, arrays and foreign keys associated with each entity. “Using the Data Dictionary” on page 144 discusses the standard *Data Dictionary* view in more detail.
- Use the *migration view* to assist you in converting data from a legacy application. This view provides a subset of the information in the standard view of the application entities that is more useful for those working on the conversion of legacy data.

The Migration View of the Data Dictionary

The standard *Data Dictionary* view separates out entity subtypes from the main entity supertype. In brief, a *supertype* relates to a *subtype* in a parent-child relationship. For example, if a **Contact** data entity is the supertype, then **Person** and **Company** are examples of its subtypes. Thus, an entity subtype inherits the characteristics of its supertype and adds individual variations particular to it.

This separation into supertype and subtype is not particularly useful for data conversion (the process of importing data into PolicyCenter from an external legacy application). Therefore, the migration view of the *Data Dictionary* differs from the standard view in the following respects:

1. The migration view displays subtype fields interspersed with supertype fields. For example:
 - `fieldA`
 - `fieldB` (only for subtype XYX)
 - `fieldC` (only for subtype DFG)
 - `fieldD`
2. The migration view does not show virtual fields or virtual arrays.
3. The migration view does not show non-loadable columns. For example, it does not show `createUserID` or `createTime`.
4. The migration view omits any non-persistent entities.
5. The migration view omits entities that are persistent but non-loadable. For example, **Group** is not loadable. Therefore, the migration view does not display it.

Using the Data Dictionary

You use the *Data Dictionary* to do the following:

- To determine what a field means that you see in a data view definition.

- To see what fields are available to add to a view, or to use in rules, or to export in an integration template, and more.
- To view the list of options for an associated typekey field. (See “What is a Typelist?” on page 266 for information on typelists.)

You navigate the dictionary like a web site, with links leading you to associated pages. You can use the **Back** and **Forward** controls of your browser to take you to previously visited pages. Within the *Data Dictionary*, you have the option to navigate to the **Data model** or the **Typelists** views. If you click **Data model**, PolicyCenter displays a left-side pane listing all of the entities in PolicyCenter. Then, on the right-side, PolicyCenter displays a pane that shows the details of the selected item in the left-side pane.

Within the details of an object, you can follow links to related objects or view the allowed values for a typelist.

The following topics describe:

- Field Colors
- Object Attributes
- Entity Subtypes
- Data Column and Field Types
- Virtual Properties on Data Entities

Field Colors

An examination of the *Data Dictionary* shows fields in green, blue, and red. These colors have the following meanings:

Color	Meaning
Green	<p>The object field (column) is part of the Guidewire base configuration. The object definition file exists in Studio in the following locations:</p> <ul style="list-style-type: none">• config → configuration → Metadata• config → configuration → Extensions
Blue	<p>The object field (column) is defined in an extension file, either by Guidewire or as a user customization. The object definition file exists in Studio in the following location:</p> <ul style="list-style-type: none">• config → configuration → Extensions <p>It is possible for Guidewire to define a base object in the Metadata folder, and then to extend the object using an extension entity in the extensions folder.</p>
Red	<p>Occasionally, it is possible to see a message in red in the Data Dictionary that states:</p> <p><i>This entity is overwritten by the application during staging.</i></p> <p>This message indicates that Guidewire PolicyCenter auto-populates a table or column's staging table equivalent. Do not attempt to populate the table yourself as the loader import process overwrites the staging table during import.</p> <p>See also the description of the <code>overwrittenInStagingTable</code> attribute in “Entity Data Objects” on page 164.</p>

Object Attributes

An object in the PolicyCenter data model can have a number of special attributes assigned to it. These attributes describe the object (or entity) further. You use the *Data Dictionary* to see what these are. For example, the **Policy** entity has the attributes **Editable**, **Exportable**, **Extendable**, **Final**, **Keyed**, **Loadable**, **Sourceable**, and **Versionable**.

The following list describes the possible attributes:

Attribute	Description
Abstract	The entity is a supertype. However, all instances of it must be one of its subtypes. That is, you cannot instantiate the supertype entity itself. An abstract entity is appropriate if the supertype serves only to collect logic or common fields, but does not make sense to exist on its own.
Editable	The related database table contains rows that you can edit. An Editable table manages additional fields that track the immediate status of an entity in the table. For example, it tracks who created it and the time, and who last edited it and the time.
Extendable	It is possible to extend the entity with additional custom fields added to it.
Final	It is not possible to subtype this entity. You can, however, extend it by adding fields to it.
Keyed	The entity has a related database table that has a primary key. Each row in a Keyed table has an integer primary key named ID. PolicyCenter manages these IDs internally, and the application ensures that no two rows in a keyed table have the same ID. You can also associate an external unique identifier with each row in a table.
Loadable	It is possible to load the entity through the use of staging tables.
Sourceable	The entity links to an external source. Each row in a table for a Sourceable entity has additional fields to identify the external application and store the ID of the Sourceable entity in the external application.
Supertype	The entity has a single table that represents multiple types of entities, called subtypes. Each subtype shares application logic and a majority of its fields. Each subtype can also define fields that are particular to it.
Temporary	The entity is a temporary entity created as part of an upgrade or staging table loading. PolicyCenter deletes the entity after the operation is complete.
Versionable	The entity has a version number that increases every time the entity changes. The PolicyCenter cache uses the version number to determine if updates have been made to an entity.

To view the definition of a particular attribute, click the tiny question mark (?) by the attribute name in the attribute list in the *Guidewire Data Dictionary*.

Entity Subtypes

If you look at Contact in the *Guidewire Data Dictionary*, for example, you see that data dictionary lists a number of subtypes. For certain PolicyCenter objects, you can think of the object in several different ways:

- As a generic object. That is, all contacts are similar in many ways.
- As a specific version or subtype of that object. For example, you would want to capture and display different information about companies than about people.

PolicyCenter creates Contact object subtypes by having a base set of shared fields common to all contacts and then extra fields that exist only for the subtype.

PolicyCenter also looks at the subtype as it decides which fields to show in the PolicyCenter interface. You can check which subtype a contact is by looking at its subtype field (for example, in a Gosu rule or class).

Data Column and Field Types

You can use the *Data Dictionary* to view the type of each object field. The following list describes some of the possible field types on an object:

Type	Description
array	Represents a one-to-many relationship, for example, contact to addresses. There is no actual column in the database table that maps to the array. PolicyCenter stores this information in the metadata.
column	As the name specifies, it indicates a column in the database.
foreign key	References a keyable entity. For example, Policy has a foreign key (AccountID) to the related account on the policy, found in the Account entity.

Type	Description
typekey	Represents a discrete value picked from a particular list, called a typelist.
virtual property	Indicates a derived property. PolicyCenter does not store virtual properties in the PolicyCenter physical database.

Virtual Properties on Data Entities

The *Data Dictionary* lists certain entity properties as *virtual*. PolicyCenter does not store virtual properties in the PolicyCenter physical database. Instead, it derives a virtual property through a method, a concatenation of other fields, or from a pointer (foreign key) to a field that resides elsewhere.

For example, if you view the Account entity in the *Data Dictionary* (for PolicyCenter), you see the following next to the **AccountContactRoleSubtypes** field:

```
Derived property returning gw.api.database.IQueryResult (virtual property)
```

Examples

The following examples illustrate some of the various ways that Guidewire applications determine a virtual property. The following examples use Guidewire ClaimCenter for illustration.

Virtual Property Based on a ForeignKey

`Claim.BenefitsDecisionReason` is a virtual property that simply pulls its value from the `cc_claimtext` table, which stores `ClaimText.ClaimTextType = BenefitsDecisionReason`. It returns a `mediumtext` value. The other fields in `cc_claimtext` and `cc_exposuretext` work in a similar fashion.

Virtual Property Based on an Associated Role

`Claim.claimant` is a virtual property that retrieves the `Contact` associated with the `Claim` with the `ClaimContactRole` of `claimant`. It returns a `Person` value.

Virtual Property Based on a Typelist

`Contact.PrimaryPhoneValue` is a virtual property that calculates its return value based on the value from `Contact.PrimaryPhone`. It retrieves the telephone number stored in the field represented by that `typekey`. This can be one of the following:

- `Contact.HomePhone`
- `Contact.WorkPhone`
- `Person.CellPhone`

It returns a phone value.

The PolicyCenter Data Model

The in PolicyCenter *data model* comprises the persistent data objects, called *entities*, that PolicyCenter manages in the application database.

This topic includes:

- “What is the Data Model?” on page 149
- “Overview of Data Entities” on page 151
- “Base PolicyCenter Data Objects” on page 160
- “Data Object Subelements” on page 177

What is the Data Model?

At its simplest, the Guidewire data model is a set of XML-formatted metadata definitions of entities and type-lists.

Entities	An <i>entity</i> defines a set of fields for information. You can add the following kinds of fields to an entity: <ul style="list-style-type: none">• Column• Type key• Array• Foreign key• Edge foreign key
Typelists	A <i>typelist</i> defines a set code/value pairs, called <i>typecodes</i> , that you can specify as the allowable values for the type key fields of entities. Several levels of restriction control what you can modify in typelists: <ul style="list-style-type: none">• Internal typelists – You cannot modify internal typelists because the application depends upon them for internal application logic.• Extendable typelists – You can modify this kind of typelist according to its schema definition.• Custom typelists – You can also create custom typelists for use on new fields on existing entities or for use with new entities.

Guidewire PolicyCenter loads the metadata of the data model on start-up. The loaded metadata instantiates the data model as a collection of tables in the application database. Also, the loaded metadata injects Java and Gosu classes in the application server to provide a programmatic interface to the entities and typelists in the database.

The Data Model in Guidewire Application Architecture

Guidewire applications employ a metadata approach to data objects. PolicyCenter uses metadata about application domain objects to drive both database persistence objects and the Gosu and Java interfaces to these objects.

This architecture provides enormous power to extend Guidewire application capabilities. Typically, you alter enterprise-level software applications through customization, wherein you change the behavior of the software by editing the code itself. In contrast, a Guidewire application uses XML files that provide default behavior, permissions and objects in the base configuration. You change the behavior of the application by modifying the base XML files and by creating Gosu business rules, classes, enhancements, and other objects.

The Base Data Model

The PolicyCenter data model specifies the entities, fields, and other definitions that comprise a default installation of PolicyCenter.

For example, the PolicyCenter data model defines a `PolicyPeriod` entity and several fields on it, such as `AccountNumber`, `IssueDate`, and `Status`.

PolicyCenter lets you change its data model to accommodate your business needs. You make your changes to the data model by modifying existing XML files and adding new ones. PolicyCenter stores your files that change the data model in the following application directory:

`PolicyCenter/modules/configuration/config/extensions`

However, you always access and edit the data model files indirectly through the `configuration → config → Extensions` folder in Studio. Do not edit the XML files directly from the file system yourself.

Guidewire calls changes that you make to the data model *data model extensions*. For example, you can extend the data model by adding new fields to the `User` entity, or you can declare entirely new entities. The complete data model of your PolicyCenter installation comprises the PolicyCenter model and any data model extensions that you make.

WARNING Do not attempt to modify any files other than those in the `PolicyCenter/modules/configuration` directory. Any attempt to modify files outside of this directory can prevent the PolicyCenter application from starting.

Working with Dot Notation

Many places within PolicyCenter require knowledge of fields within the application data model, especially while you configure PolicyCenter. For example, code in a business rule, class or enhancement may need to check the *owner* of an assignable object. Or, code may need to check the date and time of object creation. PolicyCenter provides an easy and consistent method of referring to fields within the data model, using relative references based on a *root object*.

A root object is the starting point for any field reference. If you run Gosu submission rules on a policy for example, the policy is the root object and you can access anything that relates to this policy. On the other hand, if you run Gosu assignment code for an activity, the activity is the root object. In this case, you have access to fields that relate to the activity, including the `User` associated with the activity.

Guidewire applications use *dot notation* for relative references. For example, assume that your Guidewire application has `Account` as the root object. For a simple reference to a field on the account such as `AccountNumber`, you simply use:

`account.AccountNumber`

However, suppose that you want to reference a field on an entity that relates to the account, such as a policy expiration date. You must first describe the path from the account to the policy, then describe the path from the policy to the policy expiration date:

```
account.Policy.ExpirationDate
```

Overview of Data Entities

Data entities are the high-level business objects used by PolicyCenter, such as a `Policy`, `PolicyLine`, or `Coverage`. An entity serves as the root object for data views, rules, Gosu classes, and most other data-related areas of PolicyCenter. Guidewire defines a set of data objects in the base PolicyCenter configuration from which it derives all other objects and entities. For many of the Guidewire base entities, you can also create entity extensions that enhance the base entities and provide additions required to support your particular business needs. In some cases, you can even define entirely new entities.

Data Entity Metadata Files

You define data entities through XML elements in the entity metadata definition files. The root element of an entity definition specifies the kind of entity and any attributes that apply. Subelements of the entity element define entity components, such as columns, or fields, and foreign keys.

WARNING Do not modify any of the base data entity definition files (those in the `modules/configuration/config/metadata` directory) by editing them directly. You can view these files in read-only mode in Studio in the `configuration → config → Metadata` folder.

To better understand the syntax of entity metadata, it is sometimes helpful to look at the PolicyCenter data model and its metadata definition files. PolicyCenter uses separate metadata definition files for entity declarations and extensions to them.

The base metadata files are available in Studio in the following location: `configuration → config → Metadata`

The extension metadata files are available in Studio in the following location: `configuration → config → Extensions`

The file extensions of metadata definition files distinguish their type, purpose, and contents.

File type	Purpose	Contains	Definition type
.dti	Data Type Info	A single data type definition.	datatype
Entities			
.eti	Entity Type Information	A single Guidewire or custom entity declaration. The name of the file corresponds to the name of the entity being declared.	component delegate deleteEntity entity nonPersistentEntity subtype viewEntity
Internal Extensions			
.eix	Entity Internal eXtension	A single Guidewire entity extension. The name of the file corresponds to the name of the Guidewire entity being extended.	internalExtension
Type Extensions			
.etx	Entity Type eXtension	A single Guidewire or custom entity extension. The name of the file corresponds to the name of the entity being extended.	extension viewEntityExtension
Typelists			

File type	Purpose	Contains	Definition type
.tti	Typelist Type Info	A single Guidewire or custom typelist declaration. The name of the file corresponds to the name of the typelist being declared.	typelist
.tix	Typelist Internal eXtension	A single Guidewire typelist extension. The name of the file corresponds to the name of the Guidewire typelist being extended.	internalTypelistExtension
.txx	Typelist Type eXtension	A single Guidewire or custom typelist extension. The name of the file corresponds to the name of the typelist being extended.	typelistExtension

The type of a metadata definition file determines where you can store and whether you can modify its contents.

File type	Location	Files are modifiable
.dti	configuration/config/datatypes	No
Entities		
.eti	configuration/config/extensions/entity	Yes
	configuration/config/metadata/entity	No
.eix	configuration/config/metadata/entity	No
.etx	configuration/config/extensions/entity	Yes
Typelists		
.tti	configuration/config/extensions/typelist	Yes
	configuration/config/metadata/typelist	No
.tix	configuration/config/metadata/typelist	No
.txx	configuration/config/extensions/typelist	Yes

The Metadata Directory

The metadata directory contains the metadata definition files for entities that comprise the PolicyCenter data model.

A metadata directory contains the following metadata definition file types:

- **Declaration files** – Versions of metadata definition files with extensions *.eti and *.tti.
- **Internal extension files** – Versions of metadata definition files with extensions *.eix or *.tix.

For an example, the PolicyCenter data model includes the following metadata definition files that collectively define the Address entity type.

File version	Metadata directory	File purpose
Address.eti	configuration/config/metadata/entity	Entity definition
Address.eix	configuration/config/metadata/entity	Extension to the entity definition

At runtime, Guidewire merges the .eti and .eix versions of the Address definition file to create a complete PolicyCenter Address entity type.

The Extensions Directory

The `configuration/config/extensions` directory contains your data model definitions that extend the PolicyCenter data model. PolicyCenter considers the base definitions in `configuration/config/extensions` first, and then applies the definitions in the `extensions` directory to them. This lets you create an entity extension that overrides any Guidewire entity extensions.

Example of Activity Metadata and Extension Files

The PolicyCenter data model includes the following metadata definition files that collectively define the PolicyCenter `Activity` entity.

File	Location	Purpose
<code>Activity.eti</code>	<code>configuration/config/metadata/entity</code>	Entity definition, not modifiable.
<code>Activity.eix</code>	<code>configuration/config/metadata/entity</code>	Entity extension, not modifiable.

To extend the PolicyCenter `Activity` entity, create the following extension file through Guidewire Studio.

File	Location	Purpose
<code>Activity.etx</code>	<code>configuration/config/extensions/entity</code>	Custom entity extension.

WARNING Use only Guidewire Studio to create metadata definition files. Use of Studio assures that the files reside in the correct location.

See also

- For information on how Guidewire PolicyCenter creates merged virtual directories and the directory hierarchy in general, see “PolicyCenter Configuration Files” on page 89.

The `extensions.properties` File

In general, if you change the data model by creating custom data model extensions in directory `configuration/config/extensions`, PolicyCenter automatically upgrades the database the next time you start the application server. It detects changes to files in that directory by recording a checksum each time the application server starts. If the recorded checksum and the current checksum differ, PolicyCenter upgrades the database.

Sometimes you want to force PolicyCenter to upgrade the database without making changes to your custom data model extensions in `configuration/config/extensions`. The `configuration → config → extensions` folder in Studio contains an `extensions.properties` file that contains a numeric property, `version`. The value of this property represents the current version of the data model definition for your instance of PolicyCenter. It controls whether PolicyCenter performs a database upgrade on server startup.

Whenever PolicyCenter upgrades the database, PolicyCenter stores the value of the `version` property in the database. The next time the application server starts up, PolicyCenter compares the value of the property in the database to the value in the `extensions` file. If the value in the database is lower than the value in the file, PolicyCenter performs a database upgrade. If the value in the database is higher, the upgrade fails.

WARNING In a production environment, Guidewire requires that you increment the version number whenever you make changes to the data model before you restart the application server. Otherwise, unpredictable results can occur. Use of the `extensions.properties` file in a development environment is optional.

Working with Data Entity Definition Files

In working with data entity definition files, you typically want to perform the following operations:

- Search for an existing entity definition
- Create a new entity definition
- Extend an existing entity definition

This section describes procedures for each operation.

Note: Guidewire strongly recommends that you verify your data entity definitions at the time that you create them. To do so, right-click the entity in the Project window, and then click **Validate**. The verification process highlights any issues with a data model definition, enabling you to correct any issue in a timely fashion.

Search for an Existing Entity Definition

1. In the Project window, press **Ctrl+N**.

The **Enter class name** dialog opens.

2. Type the name of the entity that you want to find.

Studio displays a list of matching entries that start with the character string that you typed.

3. In the list, click the name of the entity definition that you want to view.

Pay attention to the class type. For example, if you type “Activity”, Studio displays a list that includes all components whose name contains that text. Look for the one that says **(entity)** after it.

Result

Studio opens the file in its editor.

Create a New Entity Definition

1. In the Project window, navigate to **configuration** → **config** → **Extensions** → **Entity**.

2. Right-click **Entity**, and then click **New** → **Entity**.

3. In the **Entity** text box, type the name of the new entity definition that you want to create. Set the other properties for the entity.

4. Click **OK**.

Result

Studio displays the name of your new file in the **Extensions** → **entity** folder in Studio, and it stores the new file in the file system at the following location.

`configuration/config/extensions/entity`

Then, Studio opens your new file in its editor.

Extend an Existing Entity Definition

You can extend only entity definition files that have the **.eti** extension.

To extend an existing entity definition

1. In the Project window, navigate to **configuration** → **config** → **Metadata**, and then expand **Entity**.

2. Right-click the entity that you want to extend, and then click **New** → **Entity Extension**.

The file that you want to extend must have the **.eti** extension.

3. In the Entity Extension dialog, Studio displays the name and location of the extension file it will create. Click OK.

Result

Studio displays the name of your new file in the Extensions → entity folder and stores the new file at the following location.

```
configuration/config/extensions/entity
```

Studio then opens your new file in its editor.

PolicyCenter Data Entities

PolicyCenter uses XML metadata files to define all data entities in the data model. The datamodel.xsd file defines the elements and attributes that you can include in the XML metadata files. You can view a read-only version of this file in the configuration → xsd → metadata folder in Studio.

WARNING Do not attempt to modify datamodel.xsd. You can invalidate your PolicyCenter installation and prevent it from starting thereafter.

File datamodel.xsd defines the following:

- The set of allowable or valid data entities
- The attributes associated with each data entity
- The allowable subelements on each data entity

All PolicyCenter entity definition files must correspond to the definitions in datamodel.xsd.

Using XML files, Guidewire defines a data entity as a root element in an XML file that bears the name of the entity. For example, Guidewire declares the Activity entity type with the following Activity.eti file:

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
        desc="An activity is a instance of work assigned to a user and belonging to a claim."
        entity="Activity"
        exportable="true"
        extendable="true"
        javaClass="com.guidewire.pl.domain.activity.ActivityBase"
        platform="true"
        table="activity"
        type="retireable">
    ...
</entity>
```

At application server start up, PolicyCenter loads the XML definitions of the data entities into the application database.

Data Entities and the Application Database

Guidewire defines each data entity as a root XML element in the file that bears its name. For example, Guidewire defines the Activity data entity in Activity.eti:

```
<entity xmlns="http://guidewire.com/datamodel"
        entity="Activity"
        ...
        type="retireable">
    ...
</entity>
```

Notice that for the base configuration Activity object, Guidewire sets the type attribute to retireable. The type attribute that determines how PolicyCenter manages the data entity in the PolicyCenter database. For example:

- If a data entity has a type value of versionable, PolicyCenter stores instances of the entity in the database with a specific ID and version number.

- If a data entity has a type value of `retireable`, PolicyCenter stores instances of the entity in the database forever. However, you can *retire*, or hide, specific instances so that PolicyCenter does not display them in the interface.

IMPORTANT For each data entity in the PolicyCenter data model and for each entity type that you declare, PolicyCenter automatically generates a field named `ID` that is of data type `key`. An `ID` field is the internally managed primary key for the object. Do not attempt to create entity fields of type `key`. The `key` type is for Guidewire internal use only. Guidewire also reserves the exclusive use of the following additional data types: `foreignkey`, `typekey`, and `typeListkey`.

The following table lists the possible values for the entity type attribute. Use only those type attributes marked for general use to create or extend an data entity. Do not attempt to create or extend an entity with a type attribute marked for internal-use.

Type attribute	Usage	Description
<code>editable</code>	Internal use	An <code>editable</code> entity is a <code>versionable</code> entity. PolicyCenter automatically stores the version number of an <code>editable</code> entity. In addition to the standard <code>versionable</code> attributes of <code>version</code> and <code>ID</code> , an <code>editable</code> entity has the following additional attributes: <ul style="list-style-type: none"> <code>CreateUser</code> and <code>CreateTime</code> <code>UpdateUser</code> and <code>UpdateTime</code> <i>Guidewire recommends that you do not use this entity type. Use <code>versionable</code> instead.</i>
<code>effdated</code>	General use	An entity that has effective date fields, meaning a start and end date, used within Guidewire PolicyCenter. An <code>effdated</code> entity is a member of an effective dated graph, rooted at an <code>effdatedbranch</code> entity. PolicyCenter auto-splits the date fields during editing, in some modes. <code>Effdated</code> extends <code>editable</code> .
<code>effdatedbranch</code>	Internal use	An entity that defines the entity type of the root of a tree that contains <code>effdated</code> entities. <code>Effdatedbranch</code> extends <code>retireable</code> . <i>Guidewire recommends that you do not attempt to create an entity with a type attribute of <code>effdatedbranch</code>.</i>
<code>effdatedcontainer</code>	Internal use	An entity that defines the entity type that has branch children. Guidewire recommends that you do not attempt to create an entity with a type attribute of <code>effdatedcontainer</code> . <i>Guidewire recommends that you do not use this entity type. Use <code>versionable</code> instead.</i>
<code>joinarray</code>	Internal use	A <code>joinarray</code> entity works in a similar manner to a <code>versionable</code> entity. <i>Guidewire recommends that you do not use this entity type. Use <code>versionable</code> instead.</i>
<code>keyable</code>	Internal use	A <code>keyable</code> entity that has an <code>ID</code> , but it is not <code>editable</code> . It is possible to delete entities of this type from the database. <i>Guidewire recommends that you do not use this entity type. Use <code>versionable</code> instead.</i>
<code>nonkeyable</code>	Internal use	An entity that does not have a key. Use this type of entity in a reference or lookup table, for example. It is possible to delete entities of this type from the database. <i>Guidewire recommends that you do not attempt to create an entity with a type attribute of <code>nonkeyable</code>.</i>

Type attribute	Usage	Description
retireable	General use	<p>The <code>retireable</code> entity is an extension of the <code>editable</code> entity, and is the most common type of entity. Most, but not all, base entities are of this type.</p> <p>After PolicyCenter adds an instance of a <code>retireable</code> data entity to the database, PolicyCenter never deletes the instance. Instead, PolicyCenter retires the instance. For example, if you select a <code>retireable</code> instance in a list view and then click Delete, PolicyCenter preserves the instance in the database. However, PolicyCenter inserts an integer in the <code>Retired</code> column for the row that represents the instance. Any non-zero value in the <code>Retired</code> column indicates that PolicyCenter considers the instance retired.</p> <p>PolicyCenter automatically creates the following fields for <code>retireable</code> entities:</p> <ul style="list-style-type: none"> • <code>ID</code> and <code>PublicID</code> • <code>CreateUser</code> and <code>CreateTime</code> • <code>UpdateUser</code> and <code>UpdateTime</code> • <code>Retired</code> • <code>BeanVersion</code> <p>These are the same fields as those PolicyCenter creates for <code>editable</code> entities, with the addition of <code>Retired</code> property.</p> <p>IMPORTANT Although it is extremely common for a base entity to be retireable, it is not required. You cannot assume this to be the case. Always check the <i>Data Dictionary</i> to determine the retireability of an entity.</p>
versionable	General use	<p>An entity that has a version and ID. Entities of this type can detect concurrent updates. In general practice, Guidewire recommends that you use this entity type instead of <code>keyable</code>. <code>Versionable</code> extends <code>keyable</code>.</p> <p>It is possible to delete entities of this type from the database.</p>

PolicyCenter Database Tables

For every entity type in the data model, PolicyCenter creates a table in the application database. For example, PolicyCenter creates a `Policy` table to store information about the `Policy` object.

In the application database, you can identify an entity or extension table by the following prefix:

<code>pc_</code>	Entity table – one for each entity in the base configuration
<code>pcx_</code>	Extension table – one for each custom extension added to the <code>extensions</code> folder in Studio

Note: It is possible to create nonpersistent entities. These are entities or objects that you cannot save to the database. Guidewire discourages the use of non-persistent entities in favor of Plain Old Gosu Objects (POGOs), instead. See “NonPersistent Entity Data Objects” on page 170 for more information.

Besides entity tables, PolicyCenter creates the following types of tables in the database:

- Shadow tables
- Staging tables
- Temporary (temp) tables

Shadow Tables

A shadow table stores a copy of data from a main table for testing purposes. Every entity table potentially has a corresponding shadow table. Shadow tables in the database have one of the following prefixes:

<code>pct_</code>	Entity shadow table
<code>pctt_</code>	Typelist shadow table

PolicyCenter creates shadow tables at server startup only if before you start the server you set the `server.running.tests` system property to `true` explicitly or programmatically.

Shadow tables provide a way to quickly save and restore test data. All GUnit tests, including those that you write yourself, use shadow tables automatically. You cannot prevent GUnit tests from using shadow tables. GUnit tests use shadow tables according to the following process

1. GUnit copies data from the main application tables to the shadow tables to create a backup your test data.
2. GUnit runs your tests.
3. GUnit copies data backed up data in shadow tables to the main tables to restore a fresh copy of your test data for subsequent tests.

Staging Tables

PolicyCenter generates a staging table for any entity that is marked with an attribute of `loadable="true"`. The `loadable` attribute is `true` by default in the base configuration. A staging table largely parallels the main entity table except that:

- PolicyCenter replaces foreign keys by `PublicID` objects of type `String`.
- PolicyCenter replaces typecode fields by `typekey` objects of type `String`.

After you load data into these staging tables, you run the command line tool `table_import` to bulk load the staging table data into the main application database tables. See “Table Import Command” on page 166 in the *System Administration Guide* for information on use this command.

IMPORTANT Some data types, for example, `Entity`, contain an `overwrittenInStagingTable` attribute. If this attribute is set to `true`, then do not attempt to populate the associated staging table yourself because the loader import process overwrites this table.

In the application database, you can identify a staging table by the following prefix `pcst_`.

Temporary (Temp) Tables

PolicyCenter generates a temporary table for any entity that is marked with an attribute of `temporary="true"`. Do not confuse a temporary table with a shadow table, they are not synonymous. PolicyCenter uses temporary tables as work tables during installation or upgrade only. PolicyCenter does not use them if the server is running in standard operation.

Unfortunately, it is easy to forget to clear up these tables if they are no longer needed. Therefore, it is quite possible for an application to have several of these temporary tables remaining even though the upgrade triggers that used them are long gone.

In the application database, temporary tables look like any other entity table except that temporary tables are almost always empty.

Data Objects and Scriptability

Guidewire defines *scriptability* as the ability of code to *set* (write) or *get* (read) a scriptable item such as a property (column) on an entity. To do so, you set the following attributes:

- `getterScriptability`
- `setterScriptability`

The following table lists the different types of scriptability:

Type	Description
all	Exposed in Gosu, wherever Gosu is valid, for example, in rules and PCF files
doesNotExist	Not exposed in Gosu
hidden	Not exposed in Gosu

If you do not specify a scriptability annotation, then PolicyCenter defaults to a scriptability of all.

IMPORTANT There are subtle differences in how PolicyCenter treats entities and fields marked as doesNotExist and hidden. However, these differences relate to internal PolicyCenter code. For your purpose, these two annotations behave in an identical manner, meaning any entity or field that uses one of these annotations does not show in Gosu code. In general, there is no need for you to use either one of these annotations.

Scriptability Behavior on Entities

If you set `setterScriptability` at the entity level but you also set the value to `hidden` or `doesNotExist`, then Guidewire does not generate constructors for the entity. In essence, you cannot create a new instance of the entity in Gosu. Within the PolicyCenter data model, you can set the following scriptability annotation on `<entity>` objects:

Object	Set (write)	Get (read)
<code><entity></code>	Yes	No ¹
1. <code><entity></code> does not contain a <code>getterScriptability</code> attribute.		

Scriptability Behavior on Fields (Columns)

If you set `setterScriptability` at the field level, then the value that you set controls the writability of the associated property in Gosu. Within the PolicyCenter data model, you can set the following scriptability annotation on fields on `<entity>` objects:

Field	Set (write)	Get (read)
<code><array></code>	Yes	Yes
<code><column></code>	Yes	Yes
<code><edgeForeignKey></code>	Yes	Yes
<code><foreignkey></code>	Yes	Yes
<code><onetooone></code>	Yes	Yes
<code><typekey></code>	Yes	Yes

Base PolicyCenter Data Objects

All PolicyCenter objects exist as one of the base data objects or as a subtype of a base object. The following table lists the data objects that Guidewire defines in the base PolicyCenter configuration.

Data object	Extension	Folder
<component>	.eti	metadata, extensions
<delegate>	.eti	metadata, extensions
<deleteEntity>	.eti	extensions
<entity>	.eti	metadata, extensions
<extension>	.etx	extensions
<nonPersistentEntity>	.eti	metadata, extensions
<subtype>	.eti	metadata, extensions
<viewEntity>	.eti	metadata
<viewEntityExtension>	.etx	extensions

IMPORTANT There is an additional data object, <internalExtension>, that Guidewire uses for internal purposes. Do not attempt to create or extend this type of data entity.

Component Data Objects

A Component data object is similar to a compound property in that it represents a group of fields that all go together. Guidewire defines this object in the data model metadata files as the <component> root XML element.

Note: PolicyCenter stores all database columns on the Component entity on the parent entity.

Example Implementation

Suppose that you define a MoneyComponent data object that represents a monetary amount. The XML definition of the <component> element includes the following subelements:

- a <column> element that represents the numeric amount
- a <typekey> element that represents the currency type

The following example illustrates the monetary amount component named MoneyComponent.

```
<component name="MoneyComponent">
  <column name="Amount" type="money"/>
  <typekey name="Currency" typelist="Currency"/>
</component>
```

Note: If you need to reference a Component object from another data object, then use the element <componentref> element to create an instance of the component. For an example of how to use the <componentref> element, see “<componentref>” on page 186.

Attributes of <component>

The <component> element contains the following attributes. A value of Internal indicates that although the attribute exists, Guidewire uses it for internal purposes only.

<component> attribute	Description	Default
javaClass	<i>Internal.</i>	None
name	<i>Required.</i>	None

Subelements on <component>

The <component> element contains the following subelements:

<component> subelement	Description
column	See “<column>” on page 181.
foreignkey	See “<foreignkey>” on page 191.
fulldescription	See “<fulldescription>” on page 194.
typekey	See “<typekey>” on page 199.

Delegate Data Objects

A Delegate data object is a reusable entity that contains an interface and a default implementation of that interface. A delegate may also add its own columns to the tables of data objects that implement the delegate. This type of delegation enables a data object to implement an interface while delegating the implementation to the delegate.

You often use a delegate so objects can share code. The delegate implements the shared code rather than each class implementing copies of common code. Thus, a delegate is an entity associated with an implemented interface that multiple parent entities can reuse.

Guidewire defines delegate data object in data model metadata files as the <delegate> XML root element. You can extend existing delegates that are marked as extendable, and you can create your own delegates.

Note: As with the Component data object, PolicyCenter stores all database columns on the Delegate entity on the parent entity.

Implementing Delegate Objects

To implement most delegate objects, you add the following to an entity definition or extension.

```
<implementsEntity name="SomeDelegate"/>
```

For example, in the base configuration, the Account entity implements the Validatable delegate by using the following:

```
<entity entity="Account" ... >
  <implementsEntity name="Validatable"/>
  ...
</entity>
```

It is possible for an entity to implement multiple delegates, just as a Gosu or Java class can implement multiple interfaces.

See also

- “<implementsEntity>” on page 194

Delegate Objects That You Cannot Implement Directly

There are some delegates that you cannot implement directly through the use of the <implementsEntity> element. They are:

- Versionable
- KeyableBean
- Editable
- Retireable
- EffDated
- EffDatedBranch
- EffDatedContainer

These are special delegates that PolicyCenter implicitly adds to an entity if you set the `type` attribute on the entity to one of these values. Therefore, do not use the `<implementsEntity>` element to specify one of these delegates. Instead, use the `type` attribute on the entity declaration. The basic syntax looks similar to the following:

```
<entity name="SomeEntity" ... type="SomeDelegate">
```

For example, in the base configuration, the `Account` entity also implements the `Retirable` delegate by setting the entity `type` attribute to `retireable`.

```
<entity entity="Account" ... type="retireable">
  <implementsEntity name="Validatable"/>
  ...
</entity>
```

Also, it is not possible to explicitly implement the `EventAware` delegate. PolicyCenter automatically adds this delegate to any entity that contains an `<events>` element.

See also

- For an example of how to create a delegate object, see “Creating a New Delegate Object” on page 218.
- For a discussion of working with delegates in Gosu classes, see “Using Gosu Composition” on page 217 in the *Gosu Reference Guide*.

Attributes of `<delegate>`

The `<delegate>` element contains the following attributes.

IMPORTANT The `requires` attribute on `<delegate>` is strongly associated with the `adapter` attribute on `<implementsEntity>`. See that element discussion for details.

<code><delegate></code> attribute	Description	Default
<code>base</code>	<i>Internal.</i>	<code>false</code>
<code>effdatedOnly</code>	If true, then this delegate can only be used on effdated entities.	<code>false</code>
<code>extendable</code>	<i>Internal.</i>	<code>false</code>
<code>javaClass</code>	<i>Internal.</i> The Java class that provides an implementation of the interface.	<code>None</code>
<code>name</code>	<i>Required.</i>	<code>None</code>
<code>requires</code>	<p><i>Optional.</i> Specifies an interface for which the <i>implementers</i> of this delegate must provide an implementation. By implementers, Guidewire means those entities that specify the delegate using <code><implementsEntity></code>.</p> <p>IMPORTANT This attribute is inter-related with the <code>adapter</code> attributes of <code><implementsEntity></code>.</p> <ul style="list-style-type: none"> • If you specify a value for the <code>requires</code> attribute, then the implementers of this delegate must specify a value for the <code>adapter</code> attribute on <code><implementsEntity></code>. The value of the <code>adapter</code> attribute must be the name of a type that implements the interface specified by the <code>requires</code> attribute of the associated delegate. • If you do not specify a value for the <code>requires</code> attribute, then the implementers must not specify an <code>adapter</code> attribute on <code><implementsEntity></code>. 	<code>None</code>

Subelements of `<delegate>`

The `<delegate>` element contains the following subelements.

<code><delegate></code> subelement	Description
<code>column</code>	See “ <code><column></code> ” on page 181.
<code>datetimeordering</code>	<i>Internal.</i>
<code>foreignkey</code>	See “ <code><foreignkey></code> ” on page 191.

<delegate> subelement	Description
fulldescription	See “<fulldescription>” on page 194.
implementsEntity	See “<implementsEntity>” on page 194.
implementsInterface	See “<implementsInterface>” on page 195.
index	See “<index>” on page 195.
param	A parameter to pass as an argument to a delegate. It contains the following attributes: <ul style="list-style-type: none"> • name (use = required) • required (default = false)
typekey	See “<typekey>” on page 199.

Guidewire Recommendations

Guidewire recommends that you use delegates in the following scenarios:

- Implementing a Common Interface
- Subtyping Without Single-Table Inheritance
- Using Entity Polymorphism

Implementing a Common Interface

Guidewire recommends that you use a delegate if you want *both* of the following:

- If you want to have multiple entities implement the same interface
- If you want most of the implementations of the interface to be common

Guidewire defines a number of delegates in the base configuration, for example:

- Assignable
- Modifiable
- Validatable
- ...

To determine the list of base configuration delegate entities, search the `metadata` file folder for files that contain the following text:

```
<?xml version="1.0"?>
<delegate xmlns="http://guidewire.com/datamodel"
  ...
  ...>
```

Subtyping Without Single-Table Inheritance

Guidewire recommends that you create a delegate entity rather than define a supertype entity if you do not want to store subtype data in a single table. PolicyCenter stores information on all subtypes of a supertype entity in a single table. This can create a table that is extremely large and extremely wide. This is true especially if you have an entity hierarchy with a number of different subtypes that each have their own columns. Using a delegate avoids this single-table inheritance while preserving the ability to define the fields and behavior common to all the subtypes in one place.

Guidewire recommends that you consider carefully before making a decision on how to model your entity hierarchy.

Using Entity Polymorphism

Guidewire recommends that you create a delegate entity if you want to use polymorphism on class methods. For core PolicyCenter classes defined in Java, you cannot override these class methods on its Gosu subtypes. You can, however, push all methods and behaviors that can possibly be polymorphic into an interface, rather than the Java superclass. You can then require that all implementers of the delegate implement that interface (the `<implementsEntity>`) through the use of the delegate `requires` attribute. This delegate usage permits the use of polymorphism and enables delegate implementations to share common implementations on a common super-class.

Delete Entity Data Objects

You use the `deleteEntity` data object to remove a base configuration extension entity from the PolicyCenter data model. Guidewire defines this object in the data model metadata files as the `<deleteEntity>` XML root element.

Attributes of `<deleteEntity>`

The `<deleteEntity>` element contains the following attributes.

<code><deleteEntity></code> attribute	Description	Default
<code>name</code>	<i>Required.</i> The name of the base extension entity to delete.	None

See also

- “Removing a Base Extension Entity” on page 228

Entity Data Objects

An `Entity` data object is the standard persistent data object that Guidewire uses to define many—if not most—of the PolicyCenter entities. Guidewire defines this object in the data model metadata files as the `<entity>` XML root element.

Note: Guidewire strongly recommends that you verify your data object definitions at the time you create them. To do so, select the **Data Model Extensions** node in the **Resources** pane, right-click, and select **Verify Path**. You must start from the parent node, because the data definitions are dependent on each other. The verification process highlights any issues with a data model definition, enabling you to correct any issue in a timely fashion.

Attributes of <entity>

The <entity> element contains the following attributes.

<entity> attribute	Description	Default
abstract	If true, you cannot create an instance of the entity type at runtime. Instead, you must declare a subtype entity with abstract=false, which you can instantiate. Any of the generated code is abstract.	false
admin	Determines whether you can reference the entity from staging tables: <ul style="list-style-type: none"> Entity X has admin="true". Suppose that you have another, loadable table Y that has a foreign key to X. Then at the time you load the staging table for Y, you can load public IDs that specify entities of type X that are already in the main tables. Entity X has admin="false". Any Y that you load into a staging table must specify an X that is being loaded into the staging table for X at the same time. This is important because it allows the staging table loader to do less checking at load time. For example: <ul style="list-style-type: none"> If admin="false", then the staging table loader merely has to check that all public IDs in ccst_y specify valid entries in ccst_x. If admin="true", then the staging table loader has to check that all public IDs in ccst_y specify a valid entry in ccst_x. It must also check that all public IDs in ccst_y specify a valid entry in cc_x, the main table. 	false
autoSplit	Use with effDated entities, whether PolicyCenter auto-splits the entity on a slice edit.	true
base	<i>Internal</i> . Do not use. The default is false. Guidewire reserves the right to remove this attribute in a future release.	false
cacheable	<i>Internal</i> . If set to false, then Guidewire prohibits entities of this type and all its subtypes from existing in the global cache.	true
consistentChildren	<i>Internal</i> . If set to true, then PolicyCenter generates a consistency check and a loader validation that tries to ensure that links between child entities of this entity are consistent. Guidewire enforces the constraint only while loading data from staging tables. You can detect violations of the constraint on data committed to entity tables after the fact by running a consistency check. IMPORTANT Guidewire does not enforce consistentChildren constraints at bundle commit.	false
desc	A description of the purpose and use of the entity.	None
displayName	<i>Optional</i> . Creates a more human-readable form of the entity name. You can access this name using the following: <code>entity.DisplayName</code> If you do not specify a value for the DisplayName attribute, then the <code>entity.DisplayName</code> method returns the value of the entity attribute, instead. If you subtype an entity that has a specified display name, then the <code>entity.DisplayName</code> method returns the name of the subtype key.	None
effDatedBranchType	Only relevant for entities of type effDated. This values defines the type of the root of the tree in which the effDated entity lives.	None
entity	<i>Required</i> . The name of the entity. You use this name to access the entity in data views, rules, and other areas within PolicyCenter.	None
exportable	<i>Deprecated</i> . Only used with RPCE web services, which are deprecated.	false
extendable	<i>Internal</i> . If true, it is possible to extend this entity.	true

<entity> attribute	Description	Default
final	If true, you cannot subtype the entity. If false, you can define subtypes using this entity as the supertype. IMPORTANT If you define this incorrectly, PolicyCenter generates an error message upon resource verification and the application server refuses to start. PolicyCenter generates this verification error: <ul style="list-style-type: none">• If you attempt to subtype an entity that is marked as final that exists in the metadata folder in Studio.• If you attempt to subtype an entity that is marked as final that exists in the extensions folder in Studio.	true
generateInternallyIfAbsent	<i>Internal.</i> Do not use.	false
ignoreForEvents	If you change (or add, or remove) an entity X that does not generate events, then PolicyCenter searches for all event-generating entity instances that specify X. If PolicyCenter finds any of these event-generating entity instances, it generates Changed events for those entity instances. To determine what entities reference a non-event-generating entity, PolicyCenter examines the foreign keys and arrays that point to the entity. However, if you set ignoreForEvents to true on an entity that references the non-event-generating entity, then PolicyCenter ignores that link as it determines what entities specify another entity. <ul style="list-style-type: none">• At the entity level, the ignoreForEvents attribute means changes and additions or removals from the entity do not cause Changed events to fire for any other entity.• At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events.	false
instrumentationTable	<i>Internal.</i>	false
javaClass	<i>Internal.</i>	None
loadable	If true, you can load the entity through staging tables.	true
lockable	<i>Internal.</i> If set to true, PolicyCenter adds a lock column (<code>lockingcolumn</code>) to the table for this entity. PolicyCenter uses this to acquire an update lock on a row. The most common use is on objects in which it is important to implement safe ordering of messages. In that case, the entity which imposes the safe ordering needs to be lockable. IMPORTANT Guidewire strongly recommends that you do not use this locking mechanism.	false
overwrittenInStagingTable	<i>Internal.</i> If true and the entity is loadable, the loader process auto-populates the staging table during import. IMPORTANT If set to true, do not attempt to populate the table yourself, because the loader import process overwrites this table.	false
platform	<i>Internal.</i> Do not use. The only real effect is to change the location in which the table appears in a data distribution report. IMPORTANT Guidewire reserves the right to remove this attribute in a future release.	false
priority	The priority of the corresponding subtype key. This value is only meaningful for entities participating in a subtype hierarchy, which can be either the <subtype> entities or the root <entity>.	-1

<entity> attribute	Description	Default
readOnly	<p><i>Optional.</i> The typical use of read-only entities is for tables of reference data that you import as administrative data and then never touch again.</p> <p>You can only add a read-only entity to a bundle that has the <code>allowReadOnlyBeanChanges()</code> flag set on its commit options. That means that inserting, modifying or deleting a read-only entity requires one of these special bundles.</p> <p>You cannot set bundle commit options from Gosu. Therefore, you cannot modify these entities from Gosu, unless some Gosu-accessible interface gives you a special bundle. The administrative XML import tools use such a special bundle. However, Guidewire only uses these tools internally in the PolicyCenter product model.</p>	false
setterScriptability	See “Data Objects and Scriptability” on page 158 for information.	all
size	<i>Internal.</i> The size of the database table that contains this entity.	Large
table	<p><i>Required.</i> The name of the database table in which PolicyCenter stores the data for this entity. PolicyCenter automatically prefixes table names with <code>pc_</code> for base entities and <code>pcx_</code> for extension entities.</p> <p>Guidewire recommends the following table naming conventions:</p> <ul style="list-style-type: none"> Do not begin the table name with any product-specific extension. Use all lower-case letters. Use letters only. <p>Guidewire enforces the following restrictions on the maximum allowable length of the table name:</p> <ul style="list-style-type: none"> <code>loadable="true"</code> — maximum of 25 characters <code>loadable="false"</code> — maximum of 26 characters 	None
temporary	<p><i>Internal.</i> If <code>true</code>, then this table is a temporary table that PolicyCenter uses only during installation or upgrade.</p> <p>PolicyCenter deletes all temporary tables after it completes the installation or the upgrade.</p>	false
type	<i>Required.</i> See “Overview of Data Entities” on page 151 for a discussion of data entity types.	None
typelistTableName	<p>If you create a non-final entity, then ClaimCenter automatically creates a typelist to keep track of the subtypes of that entity. That typelist has an associated database table. If you do not specify a value for this attribute, then ClaimCenter uses the name of the entity as the table name for the subtype typelist.</p> <p>However, ClaimCenter places a restriction of 25 characters on the length of the database table name. You use this attribute to specify the database table name for the typelist if an entity name is too long to become a valid typelist table name.</p> <p>It is not valid to use this attribute with entity types marked as final.</p>	None
validateOnCommit	<i>Internal.</i> Do not use. If <code>true</code> , PolicyCenter validates this entity during a commit of a bundle that contains this entity.	true

Subelements of <entity>

The `<entity>` element contains the following subelements.

<entity> subelement	Description
array	See “<array>” on page 179.
aspect	<i>Internal.</i>
checkconstraint	<i>Internal.</i>
column	See “<column>” on page 181.
componentref	See “<componentref>” on page 186.

<entity> subelement	Description
customconsistencycheck	<i>Internal.</i>
datetimeordering	<i>Internal.</i>
dbcheckbuilder	<i>Internal.</i>
edgeForeignKey	See “<edgeForeignKey>” on page 187.
events	See “<events>” on page 190.
foreignkey	See “<foreignkey>” on page 191.
fulldescription	See “<fulldescription>” on page 194.
implementsEntity	See “<implementsEntity>” on page 194.
implementsInterface	See “<implementsInterface>” on page 195.
index	See “<index>” on page 195.
jointableconsistencycheck	<i>Internal.</i>
onetoone	See “<onetoone>” on page 197.
remove-index	See “<remove-index>” on page 198.
searchColumn	See “The <searchColumn> Subelement” on page 168
tableAugmenter	<i>Internal.</i>
typekey	See “<typekey>” on page 199.
validatetypekeyinset	<i>Internal.</i>
validatetypekeynotinset	<i>Internal.</i>

The <searchColumn> Subelement

The <searchColumn> subelement on <entity> defines a search denormalization column in the database. The denormalization copies the value of a column on another table into a column on the denormalizing table. You must link the tables through a foreign key. The purpose of this denormalization is to avoid costly joins in performance-critical searches.

The use of search denormalization columns adds overhead to updates, as does any denormalization. Guidewire recommends that you only use these columns if there is an identifiable performance problem with a search that is directly related to the join between the two tables.

Note: It is possible to have a <searchColumn> sublement on the <extension> and <subtype> elements as well.

The <searchColumn> element contains the following attributes.

<searchColumn> attribute	Description	Default
columnName	Name to use for the database column corresponding to this property. If you do not specify a value, then PolicyCenter uses the name value instead.	None
deprecated	If true, then PolicyCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.	false
	If you deprecate an item, use the description to explain why. For more information, see “The deprecated Attribute” on page 178.	
desc	Description of the intended purpose of this column.	None
name	<i>Required.</i> Name of the column on the table and the field on the entity. The name value maps to the accessor and mutator methods of a field on the entity, not the actual private member field. For example, name maps to setName and getName, not the private _name member field. Column names must contain letters only. A column name cannot contain an underscore.	None

<searchColumn> attribute	Description	Default
sourceColumn	<i>Required.</i> Name of the column on the source entity, whose value this column copies. The sourceColumn must not name a localized column.	None
sourceForeignKey	<i>Required.</i> Name of a foreign key field on this entity, which refers to the source entity for this search denormalization column. The sourceForeignKey must not be importable against existing objects.	None
sourceSubtype	Optional name of the particular subtype on which the source column is defined. If not specified, then PolicyCenter assumes that the source column to exist on the entity referred to by the source object. However, you must specify this value if the sourceColumn is on a subtype of the entity referred to by sourceForeignKey.	None

For example, suppose that you set the following attribute definitions:

- `searchColumn` – *MyDenormColumn*
- `sourceForeignKey` – *Source*
- `sourceColumn` – *SourceField*

This declaration says:

Copy the value of *SourceField* on the object pointed to by the foreign key named *Source* into the field named *MyDenormColumn*. PolicyCenter automatically populates the column as part of bundle commit, staging table load, and database upgrade.

If you need to denormalize a field on a subtype of the entity referred to by the foreign key, then you can specify the optional `sourceSubtype` attribute.

As with linguistic denormalization columns, you cannot access the value of these search denormalization columns in memory. The value is only available in the database. Thus, you can only access the value through a database query.

It is possible to make a query against a search denormalization column that is a denormalization of a linguistic denormalization column. In that case, the query generator knows not to wrap the column values in the linguistic denormalization function. This preserves the optimization that linguistic denormalization columns provide.

It is important to understand that search denormalization columns specify one column only — the column that you specify with the `sourceColumn` attribute. So, if you want to denormalize both a column and its linguistic denormalization, then you need two separate search denormalization columns. However, in this case, you typically would just want to denormalize the linguistic denormalization column. You would only want to denormalize the source column if you wanted to support case-sensitive searches on it.

Search denormalization columns can only specify `<column>` or `<typekey>` fields.

Extension Data Objects

An Extension data object is the standard data object that you use to extend an already existing data object or entity. Guidewire defines this object in the data model metadata files as the `<extension>` XML root element.

See also

- For information on how to extend the base data objects, see “Modifying the Base Data Model” on page 211.

Attributes of <extension>

The <extension> element contains the following attributes.

<extension> attribute	Description	Default
entityName	<p><i>Required.</i> This value must match the file name of the entity that it extends.</p> <p>IMPORTANT PolicyCenter generates an error at resource verification if the value set that you set for the entityName attribute for an extension does not match the file name.</p>	None

Subelements of <extension>

The <extension> element contains the following subelements.

<extension> subelement	Description
array	See “<array>” on page 179.
array-override	Use to override, or flip, the value of the triggersValidation attribute of an <array> element definition on a base data object. See “Working with Attribute Overrides” on page 216 for details.
column	See “<column>” on page 181.
column-override	Use to override certain very specific attributes of a base data object. See “Working with Attribute Overrides” on page 216 for details.
componentref	See “<componentref>” on page 186.
description	A description of the purpose and use of the entity.
edgeForeignKey	See “<edgeForeignKey>” on page 187.
foreignkey	See “<foreignkey>” on page 191.
foreignkey-override	Use to override, or flip, the value of the triggersValidation attribute of a <foreignkey> element definition on a base data object. See “Working with Attribute Overrides” on page 216 for details.
implementsEntity	See “<implementsEntity>” on page 194.
implementsInterface	See “<implementsInterface>” on page 195.
index	See “<index>” on page 195.
internalonlyfields	<i>Internal.</i>
onetoone	See “<onetoone>” on page 197.
onetoone-override	Use to override, or flip, the value of the triggersValidation attribute of an <onetoone> element definition on a base data object. See “Working with Attribute Overrides” on page 216 for details.
remove-index	See “<remove-index>” on page 198.
searchColumn	See “The <searchColumn> Subelement” on page 168
typekey	See “<typekey>” on page 199.
typekey-override	Use to override certain specific attributes, or fields, of a <typekey> element definition on a base data object. See “Working with Attribute Overrides” on page 216 for details.

NonPersistent Entity Data Objects

A NonPersistentEntity data object defines a temporary, or nonpersistent, entity that PolicyCenter creates and uses only during the time that the PolicyCenter server is running. If the server shuts down, PolicyCenter discards the entity data. It is not possible to commit a NonPersistentEntity object to the database.

Guidewire defines this object in the data model metadata files as the <nonPersistentEntity> XML root element.

Note: You cannot extend a persistent entity with a nonpersistent entity.

Guidewire Recommendations for NonPersistent Entities

Guidewire recommends that you do not create or extend nonpersistent entities as a general rule. In general, do not use nonpersistent entities to obtain some desired behavior. A major issue with nonpersistent entities is that they do not interact well with data bundles. Passing a nonpersistent entity to a PCF page, for example, is generally a bad idea because it generally does not work in the manner that you expect.

The nonpersistent entity has to live in a bundle and can only live in *one* bundle. Therefore, passing it to one context removes it from the other context. Even worse, it is possible that in passing the nonpersistent entity from one context to another, the entity loses any nested arrays or links associated with it. Thus, it is possible to lose parts of the entity graph as the nonpersistent entity moves around. Entity serialization is also less efficient and less controllable than using a custom class that contains only the data that it really needs.

Guidewire recommends, therefore, that you use a Gosu class in situations in which you want the behavior of a nonpersistent entity. For example:

- If you want the behavior of a nonpersistent entity in web services, do not use a nonpersistent entity. Instead, Guidewire recommends that you create a Gosu class and then expose that as a web service rather than relying on nonpersistent entities and entity serialization.
- If you want a field that behaves, for example, as `nonnegativeinteger` column, do not use a nonpersistent entity. Instead, as you can specify a data type through the use of annotations, add the wanted data type behavior to properties on Gosu classes. See “Defining a Data Type for a Property” on page 235 for information on how to associates data types with object properties using the annotation syntax.

Attributes of `<nonPersistentEntity>`

The `<nonPersistentEntity>` element contains the following attributes.

<code><nonPersistentEntity></code> attribute	Description	Default
<code>abstract</code>	If true, you cannot create an instance of the entity type at runtime. Instead, you must declare a subtype entity with <code>abstract=false</code> , which you can instantiate. Any of the generated code is abstract.	false
<code>desc</code>	A description of the purpose and use of the entity.	None
<code>displayName</code>	<i>Optional.</i> Creates a more human-readable form of the entity name. You can access this name using the following: <code>entity.DisplayName</code> If you do not specify a value for the <code>DisplayName</code> attribute, then the <code>entity.DisplayName</code> method returns the value of the <code>entity</code> attribute, instead. If you subtype an entity that has a specified display name, then the <code>entity.DisplayName</code> method returns the name of the subtype key.	None
<code>entity</code>	<i>Required.</i> The name of the entity. You use this name to access the entity in data views, rules, and other areas within PolicyCenter.	None
<code>exportable</code>	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	false
<code>extendable</code>	If true, it is possible to extend this entity.	true
<code>final</code>	If true, you cannot subtype the entity. If false, you can define subtypes using this entity as the supertype.	true
<code>javaClass</code>	<i>Internal.</i>	None

<nonPersistentEntity> attribute	Description	Default
priority	The priority of the corresponding subtype key. This value is only meaningful for entities participating in a subtype hierarchy, which can be either the <subtype> entities or the root <entity>.	-1
typelistTableName	If you create a non-final entity, then ClaimCenter automatically creates a typelist to keep track of the subtypes of that entity. That typelist has an associated database table. If you do not specify a value for this attribute, then ClaimCenter uses the name of the entity as the table name for the subtype typelist.	None

However, ClaimCenter places a restriction of 25 characters on the length of the database table name. You use this attribute to specify the database table name for the typelist if an entity name is too long to become a valid typelist table name.

It is not valid to use this attribute with entity types marked as final.

Subelements of <nonPersistentEntity>

The <nonPersistentEntity> element contains the following subelements.

<nonPersistentEntity> subelement	Description
array	See “<array>” on page 179.
aspect	<i>Internal.</i>
column	See “<column>” on page 181.
componentref	See “<componentref>” on page 186.
edgeForeignKey	See “<edgeForeignKey>” on page 187.
foreignkey	See “<foreignkey>” on page 191.
fulldescription	See “<fulldescription>” on page 194.
implementsEntity	See “<implementsEntity>” on page 194.
implementsInterface	See “<implementsInterface>” on page 195.
onetoone	See “<onetoone>” on page 197.
typekey	See “<typekey>” on page 199.

Subtype Data Objects

A Subtype entity defines an entity that is a subtype of another entity. The subtype entity has all of the fields and elements of its supertype and it can also have additional ones. Guidewire defines this object in the data model metadata files as the <subtype> XML root element.

PolicyCenter does not associate a separate database table with a subtype. Instead, PolicyCenter stores all subtypes of a supertype in the table of the supertype and resolves the entity to the correct subtype based on the value of the Subtype field. To accommodate this, PolicyCenter stores all fields of a subtype in the database as nullable columns—even the ones defined as non-nullable. However, if you define a field as non-nullable, then the PolicyCenter metadata service enforces this for all data operations.

You can only define a subtype for any entity that has its `final` attribute set to `false`. PolicyCenter automatically creates a Subtype field for non-final entities.

Attributes of <subtype>

The <subtype> element contains the following attributes:

<subtype> attribute	Description	Default
abstract	If true, you cannot create an instance of the entity type at runtime. Instead, you must declare a subtype entity with abstract=false, which you can instantiate. Any of the generated code is abstract.	false
desc	A description of the purpose and use of the subtype.	None
displayName	<p><i>Optional.</i> Occasionally in the PolicyCenter interface, you want to display the subtype name of subtyped entity instances. Use the displayName attribute to specify a String to display as the subtype name. You can access this name using the following:</p> <pre>entity.DisplayName</pre> <p>If you do not specify a value for the displayName attribute, then PolicyCenter displays the name of the entity. The entity name is often not user-friendly. For a description of the displayName attribute, see "Entity Data Objects" on page 164.</p>	None
array	The name of the subtype entity. Use this name to access the entity in data views, rules, and other areas within PolicyCenter.	None
aspect	<p>If true, the entity definition is final and you cannot define any subtypes for it. If false, then you can define a subtype using this entity as the supertype.</p> <p>IMPORTANT If you define this incorrectly, PolicyCenter generates an error message upon resource verification and the application server refuses to start. PolicyCenter generates this verification error:</p> <ul style="list-style-type: none"> • If you attempt to subtype an entity that is marked as final that exists in the metadata folder in Studio. • If you attempt to subtype an entity that is marked as final that exists in the extensions folder in Studio. 	false
checkconstraint	<i>Internal.</i>	
column	The relative position of the subtype in a list of peer subtypes. PolicyCenter often displays the Subtype field in a supertype as a typelist. Thus, this attribute serves the same purpose as the priority attribute of a typecode in a typelist.	-1
customconsistencycheck	<i>Optional.</i>	None
datetimeordering	<p><i>Required.</i> The name of the supertype of this subtype.</p> <p>IMPORTANT If you reference a non-existent or malformed supertype name, then PolicyCenter generates an error upon resource verification and the application server refuses to start.</p>	None

Subelements of <subtype>

The <subtype> element contains the following subelements.

<subtype> subelement	Description
array	See "<array>" on page 179.
aspect	<i>Internal.</i>
checkconstraint	<i>Internal.</i>
column	See "<column>" on page 181.
customconsistencycheck	<i>Internal.</i>
datetimeordering	<i>Internal.</i>
dbcheckbuilder	<i>Internal.</i>
edgeForeignKey	See "<edgeForeignKey>" on page 187.
events	See "<events>" on page 190.

<subtype> subelement	Description
foreignkey	See “<foreignkey>” on page 191.
fulldescription	See “<fulldescription>” on page 194.
implementsEntity	See “<implementsEntity>” on page 194.
implementsInterface	See “<implementsInterface>” on page 195.
index	See “<index>” on page 195.
jointableconsistencycheck	<i>Internal.</i>
onetoone	See “<onetoone>” on page 197.
searchColumn	See “The <searchColumn> Subelement” on page 168
tableAugmenter	<i>Internal.</i>
typekey	See “<typekey>” on page 199.
validatetypekeyinset	<i>Internal.</i>
validatetypekeynotinset	<i>Internal.</i>

Subtypes and Typelists

After you define a new subtype, PolicyCenter automatically adds that entity type to the associated entity typelist. This is true, even if PolicyCenter marks that typelist as `final`.

For example, suppose that you define an `Inspector` entity as a subtype of `Person`.

```
<?xml version="1.0"?>
<subtype xmlns="http://guidewire.com/datamodel" desc="Professional inspector" displayName="Inspector"
    entity="InspectorExt"
    supertype="Person">
    <column name="InspectorLicenseExt" type="varchar" desc="Inspector's business license number">
        <columnParam name="size" value="30"/>
    </column>
</subtype>
```

Notice that while `InspectorExt` is subtype of `Person`, `Person`, itself, is a subtype of `Contact`. PolicyCenter automatically adds the new `InspectorExt` type to the `Contact` typelist. This is true, even though PolicyCenter marks the `Contact` typelist as `final`.

To see this change:

- In the *PolicyCenter Data Dictionary*, you must restart the application server.
- In the `Contact` typelist in Studio, you must restart Studio.

See also

- “Defining a Subtype” on page 223

viewEntity Data Objects

A `viewEntity` is a logical view of entity data. You can use a `viewEntity` to enhance performance during the viewing of tabular data. A `viewEntity` provides a logical view of data for an entity entities of interest to a `ListView`. A `viewEntity` can include paths from the root or primary entity to other related entities.

For example, from the `DesktopActivityView`, you can specify a column with a `Job.JobNumber` value. The `Activity` entity is the primary entity of the `DesktopActivityView`. The `Activity` entity has a corresponding `viewEntity` called `ActivityView`.

Unlike a standard `entity`, a `viewEntity` does not have an underlying database table. PolicyCenter does not persist `viewEntity` entities to the database. Instead of storing data, a `viewEntity` restricts the amount of data that a database query returns. A `viewEntity` does not represent or create a *materialized view*, which is a database table that caches the results of a database query.

Queries against a `viewEntity` type are actually run against the normal entity table in the database, as specified by the `primaryEntity` attribute of the view entity definition. The query against a `viewEntity` automatically adds any joins necessary to retrieve `viewEntity` columns if they include a bean path. However, access to `viewEntity` columns is not possible when constructing the query.

A `viewEntity` improves the performance of PolicyCenter on frequently used pages that list entities. Like other entities, you can subtype a `viewEntity`. For example, the [My Activities](#) page uses a `viewEntity`, the `ActivityDesktopView`, which is a subtype of the `ActivityView`.

Because PolicyCenter can export `viewEntity` types, it generates SOAP interfaces for them.

Note: If you create or extend a view entity that references a column that is of `type="currencyamount"`, then you must handle the view entity extension in a particular manner. See “Extending an Existing View Entity with a Currency Column” on page 227 for details.

Guidewire defines this object in the data model metadata files as the `<viewEntity>` XML root element.

Attributes of `<viewEntity>`

The `<viewEntity>` element contains the following attributes:

<code><viewEntity></code> attribute	Description	Default
<code>abstract</code>	If true, you cannot create an instance of the entity type at runtime. Instead, you must declare a subtype entity with <code>abstract=false</code> , which you can instantiate. Any of the generated code is abstract.	false
<code>desc</code>	A description of the purpose and use of the entity.	None
<code>entity</code>	<i>Required.</i> Name of this <code>viewEntity</code> object.	None
<code>exportable</code>	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
<code>extendable</code>	If true, it is possible to extend this entity.	true
<code>final</code>	If true, the entity definition is final and you cannot define any subtypes for it. If false, then you can define a subtype using this entity as the supertype.	true
<code>javaClass</code>	<i>Internal.</i>	None
<code>primaryEntity</code>	<i>Required.</i> The primary entity type for this <code>viewEntity</code> object. The primary entity must be keyable. See “Data Entities and the Application Database” on page 155 for information on keyable entities.	None
<code>showRetiredBeans</code>	Whether to show retired beans in the view.	None
<code>supertypeEntity</code>	<i>Optional.</i> The name of supertype of this entity.	None
<code>typelistTableName</code>	If you create a non-final entity, then ClaimCenter automatically creates a typelist to keep track of the subtypes of that entity. That typelist has an associated database table. If you do not specify a value for this attribute, then ClaimCenter uses the name of the entity as the table name for the subtype typelist. However, ClaimCenter places a restriction of 25 characters on the length of the database table name. You use this attribute to specify an alternate database table name for the typelist if an entity name is too long to become a valid typelist table name. It is not valid to use this attribute with entity types marked as final.	None

Subelements of <viewEntity>

The <viewEntity> elements contain the following subelements:

<viewEntity> subelement	Description
computedcolumn	Specifies a column with row values that Guidewire computes while querying the database. For example, the values of a computed column might be the sum of the values from two database columns (<code>col1 + col2</code>).
computedtypekey	Specifies a typekey that has some type of transformation applied to it during querying from the database.
fulldescription	See the discussion following the table.
viewEntityColumn	Represents a column in a viewEntity table
viewEntityLink	Uses to access another entity through a foreign key. Typically, you use this value within the PolicyCenter interface to create a link to that entity.
viewEntityName	Represents an entity name column in a viewEntity table. An entity name is a string column that contains the name of an entity that is suitable for viewing in the PolicyCenter interface.
viewEntityTypekey	Represents a typekey column in a viewEntity table.

The *Data Dictionary* uses the `fulldescription` subelement. The following example illustrates how to use this element:

```
<fulldescription>
  <! [CDATA[<p>Aggregates the information needed to display one activity row (base entity for all other
activity views).</p>]]>
</fulldescription>
```

The other subelements all require both a `name` and `path` attribute. The following code illustrates this:

```
<viewEntityName name="RelActAssignedUserName" path="RelatedActivity.AssignedUser"/>
```

Specify the `path` value relative to the `primaryEntity` on which you base the view.

The `computedcolumn` takes a required `expression` attribute and an additional, optional `function` attribute. The following is an example of a `computedcolumn`:

```
<computedcolumn name="Amount" expression="${1}" paths="LineItems.Amount" function="SUM"/>
```

The `expression` for this column can take multiple column values `${column_num}` passed from the PolicyCenter interface. For example, a valid expression is: `${1} - ${2}` with `${1}` the first column and `${2}` the second column. The `function` value must be an SQL function that you can apply to this expression. The following are legal values:

- SUM
- AVG
- COUNT
- MIN
- MAX

Note: If the SQL function aggregates data, PolicyCenter applies an SQL group automatically.

viewEntityExtension Data Objects

You use the `viewEntityExtension` entity to extend the definition of a `viewEntity` entity. Guidewire defines this object in the data model metadata files as the `<viewEntityExtension>` XML root element.

Attributes of <viewEntityExtension>

The <viewEntityExtension> element contains the following attributes:

<viewEntityExtension> attribute	Description	Default
entityName	<p><i>Required.</i> This value must match the file name of the viewEntityExtension that it extends.</p> <p>IMPORTANT PolicyCenter generates an error at resource verification if the value set that you set for the entityName attribute for a viewEntityExtension does not match the file name.</p>	None

Subelements of <viewEntityExtension>

The <viewEntityExtension> element contains the following subelements:

<viewEntityExtension> subelement	Description
computedcolumn	Specifies a column with row values that Guidewire computes while querying the database. For example, the values of a computed column might be the sum of the values from two database columns (co11 + co12).
computedtypekey	Specifies a typekey that has some type of transformation applied to it during querying from the database.
description	A description of the purpose and use of the entity.
viewEntityColumn	<p>Represents a column in a viewEntity table. The viewEntityColumn element contains a path attribute that you use to define the entity path for the column:</p> <ul style="list-style-type: none"> • The path attribute definition cannot traverse arrays. • The path attribute is always relative to the primary entity on which you base the view. <p>Note: If you reference a column of type currencyamount, you must also define the currencyProperty specified in the original column definition on the viewEntity entity. See “Extending an Existing View Entity” on page 226 for an example of this.</p>
viewEntityLink	Uses to access another entity through a foreign key. Typically, you use this value within the PolicyCenter interface to create a link to that entity.
viewEntityName	Represents an entity name column in a viewEntity table. An entity name is a string column that contains the name of an entity that is suitable for viewing in the PolicyCenter interface.
viewEntityTypekey	Represents a typekey column in a viewEntity table.

Important Caution

Guidewire strongly recommends that you not create a view entity extension—viewEntityExtension—that causes traversals into revised (effdated) data. Doing so has the possibility of returning duplicate rows if any revisioning in the traversal path splits an entity.

Instead, try one of the following:

- Denormalize the desired data onto a non-effdated entity.
- Add domain methods to the implementation of the View entity.

Data Object Subelements

This topic describes the subelements that you can use in metadata definition files. These subelements are:

- <array>
- <column>

- <componentref>
- <edgeForeignKey>
- <events>
- <foreignkey>
- <fulldescription>
- <implementsEntity>
- <implementsInterface>
- <index>
- <onetoone>
- <remove-index>
- <typekey>

Subelements for Internal Use Only

Do not use the following entity subelements. Guidewire uses these subelements for internal purposes only.

- <aspect>
- <checkconstraint>
- <customconsistencycheck>
- <datetimordering>
- <dbcheckbuilder>
- <jointableconsistencycheck>
- <tableAugmenter>
- <validatetypekeyinset>
- <validatetypekeynotinset>

The deprecated Attribute

The **deprecated** attribute applies to the following subelements:

- <array>
- <column>
- <componentref>
- <edgeForeignKey>
- <foreignkey>
- <onetoone>
- <searchColumn>
- <typekey>

The **deprecated="true"** attribute does not alter the database in any way. Instead, the **deprecated** attribute marks a data field as deprecated in the *Data Dictionary* and places a **Deprecated** annotation on the field in the *Guidewire Studio API Reference*. The **deprecated** attribute supports organizations that want to remove a field in a two-phase process.

In the first phase, you add the **deprecated** attribute to the field subelement. Studio indicates the field is deprecated whenever Gosu code references the field. During this first phase, developers work to remove the deprecated field from their code. The second phase occurs after developers remove all occurrences of the deprecated field.

In the second phase, you drop the field from the entity definition. In some cases, Guidewire will drop the column from the database automatically to synchronize the physical database with your revised data model. In most cases however, the DBA must alter the database with SQL statements run against the database to synchronize the database with your revised data model.

Guidewire generally recommends against using the `deprecated` attribute and the two-phase removal process. If you deprecate a field, Studio signals to the development team that the field is no longer used. The DBA does not receive this information. Over time, with a number of deprecated fields, the DBA manages an ever larger amount of unused information in the physical database. To avoid managing unused data, Guidewire strongly recommends that you keep your physical database and the data model of your application synchronized by dropping unused fields instead of deprecating them.

<array>

An array defines a set of additional entities of the same type to associate with the main entity. For example, a `Policy` entity includes an array of `Document` entities.

Attributes of <array>

The `<array>` element contains the following attributes:

<array> attribute	Description	Default
<code>arrayentity</code>	<i>Required.</i> The name of the entity that makes up the array.	None
<code>arrayfield</code>	<i>Optional.</i> Name of the field in the array table that is the foreign key back to this table. However, you do not need to define a value if the array entity has exactly one foreign key back to this entity. Note that even if you define only one foreign key explicitly, additional foreign keys may be created implicitly. For example, <code>CreateUserID</code> is automatically added to an editable entity. In that case, <code>arrayfield</code> would be required because there is more than one foreign key.	None
<code>cascadeDelete</code>	If true, then PolicyCenter deletes the array elements also if you delete the array container.	false
<code>deprecated</code>	If true, then PolicyCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a <code>Deprecated</code> annotation on it in the Guidewire Studio API Reference. If you deprecate an item, use the description to explain why. For more information, see “The deprecated Attribute” on page 178.	false
<code>desc</code>	A description of the purpose and use of the array.	None
<code>exportable</code>	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
<code>generateCode</code>	<i>Internal.</i>	true
<code>getterScriptability</code>	See “Data Objects and Scriptability” on page 158 for information.	all
<code>ignoreforevents</code>	If you change (or add, or remove) an entity X that does not generate events, then PolicyCenter searches for all event-generating entity instances that specify X. If PolicyCenter finds any of these event-generating entity instances, it generates <code>Changed</code> events for those entity instances. To determine what entities reference a non-event-generating entity, PolicyCenter examines the foreign keys and arrays that point to the entity. However, if you set <code>ignoreForEvents</code> to true on an entity that references the non-event-generating entity, then PolicyCenter ignores that link as it determines what entities specify another entity. <ul style="list-style-type: none"> • At the entity level, the <code>ignoreForEvents</code> attribute means changes to (or addition or removal of) this entity do not cause <code>Changed</code> events to fire for any other entity. • At the column level, the <code>ignoreForEvents</code> attribute means changes to this column do not cause the application to generate events. 	false
<code>name</code>	<i>Required.</i> The name of the property corresponding to this array	None

<array> attribute	Description	Default
owner	If true, this entity owns the objects in the array. <ul style="list-style-type: none"> • If you delete the owning object, then PolicyCenter deletes the array items as well. • If you update the contents of the array, then PolicyCenter considers the owner as updated as well. 	false
requiredmatch	One of the following values <ul style="list-style-type: none"> • all – There must be at least one matching row in the array for every row from this table. For example, there must be at least one check payee for every check. • none – There is no requirement for matching rows. • nonretired – There must be at least one matching row for every non-retired row from this table. 	None
setterScriptability	See “Data Objects and Scriptability” on page 158 for information.	all
trackssynchstate	If true, this value denotes that PolicyCenter uses the associated table to track the external synchronization state for the owning table.	false
triggersValidation	Whether changes to the entity pointed to by this array trigger validation. Changes to the array that trigger validation include: <ul style="list-style-type: none"> • The addition of an object to the array • The removal of an object from the array • The modification of an object in the array See the discussion on this attribute that follows this table.	false

If set to true, the triggersValidation attribute can trigger additional PolicyCenter processing. Exactly what happens depends on several different factors:

- If the parent entity for the array is validatable, then any modification to the array triggers the execution of the Preupdate and Validation rules on the parent entity. Validation occurs whenever PolicyCenter attempts to commit a bundle that contains the parent entity. For an entity to be validatable, it must implement the `Validatable` delegate.
- If the parent entity has preupdate rules, but no validation rules, then PolicyCenter executes the preupdate rules on the commit bundle. This is the case only if configuration parameter `UseOldStylePreUpdate` is set to true, which is the default. If `UseOldStylePreUpdate` is set to false, PolicyCenter invokes the `IPreUpdateHandler` plugin on the commit bundle instead. Then, PolicyCenter executes the logic defined in the plugin on the commit bundle.
- If the parent entity has validation rules, but no preupdate rules, then PolicyCenter executes the validation rules on the commit bundle.
- If the parent entity has neither preupdate nor validation rules then the following occurs:
 - a. In the case of `UseOldStylePreUpdate=true`, PolicyCenter does nothing.
 - b. In the case of `UseOldStylePreUpdate=false`, PolicyCenter calls the `IPreUpdateHandler` plugin on the commit bundle.

Subelements of <array>

The <array> element contains the following subelements:

<array> subelement	Description
array-association	<p>This subelement contains the following attributes:</p> <ul style="list-style-type: none">• hasContains (default = false)• hasGetter (default = true)• hasSetter (default = false)• valueField (default = ID) <p>It also contains the following subelements of its own, each of which can exist, at most, one time:</p> <ul style="list-style-type: none">• constant-map• subtype-map• typelist-map <p>See “Typelist Mapping Associative Arrays” on page 207 for more information.</p>
fulldescription	See “<fulldescription>” on page 194.
link-association	<p>This subelement contains the following attributes:</p> <ul style="list-style-type: none">• hasGetter (default = true)• hasSetter (default = false)• valueField (default = ID) <p>It also contains the following subelements of its own, each of which can exist, at most, one time:</p> <ul style="list-style-type: none">• constant-map• subtype-map• typelist-map <p>See “Subtype Mapping Associative Arrays” on page 205 for more information.</p>

<column>

The <column> element defines a single-value field in the entity.

Note: For a discussion of <column-override>, see “Working with Attribute Overrides” on page 216 for details.

Attributes of <column>

The <column> element contains the following attributes:

<column> attribute	Description	Default
columnName	<p><i>Optional.</i> If specified, PolicyCenter uses this value as the column name of the corresponding database column. If you do not specify a columnName value, then PolicyCenter uses the value of the name attribute for the database column name. The maximum value for a column name is 30 characters.</p> <p>IMPORTANT All column names on a table must be unique within that table. Otherwise, Studio displays an error if you verify the resource and the application server fails to start.</p>	None
createhistogram	<p>Whether to create a histogram on the column during an update to the database statistics.</p> <p>Note: It is possible to override this attribute on an existing column in an extension (*.etx) file using the <column-override> element. You can use the override to turn off an existing histogram or to create one that did not previously exist.</p> <p>This change does not take effect during an upgrade. The change occurs <i>only</i> if you regenerate statistics for the affected table by using the Guidewire maintenance_tools command.</p>	false
	<p>See also</p> <ul style="list-style-type: none"> • “Working with Attribute Overrides” on page 216 • “Configuring Database Statistics” on page 42 in the <i>System Administration Guide</i> • “Maintenance Tools Command” on page 159 in the <i>System Administration Guide</i> 	
default	Default value given to the field during new entity creation.	None
deprecated	<p>If true, then PolicyCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.</p> <p>If you deprecate an item, use the description to explain why.</p> <p>For more information, see “The deprecated Attribute” on page 178.</p>	false
desc	A description of the purpose and use of the field.	None
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
generateCode	<i>Internal.</i>	true
getterScriptability	See “Data Objects and Scriptability” on page 158 for information.	all
ignoreforevents	<p>If you change (or add, or remove) an entity X that does not generate events, then PolicyCenter searches for all event-generating entity instances that specify X. If PolicyCenter finds any of these event-generating entity instances, it generates Changed events for those entity instances.</p> <p>To determine what entities reference a non-event-generating entity, PolicyCenter examines the foreign keys and arrays that point to the entity. However, if you set ignoreForEvents to true on an entity that references the non-event-generating entity, then PolicyCenter ignores that link as it determines what entities specify another entity.</p> <ul style="list-style-type: none"> • At the entity level, the ignoreForEvents attribute means changes to (or addition or removal of) this entity do not cause Changed events to fire for any other entity. • At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events. 	false
loadable	If true, you can load the field through staging tables. A staging table can contain a column mapping to the field.	true

<column> attribute	Description	Default
name	<p><i>Required.</i> The name of the column on the table and the field, or property, on the entity. PolicyCenter uses this value as the column name <i>unless</i> you specify a <code>columnName</code> attribute. Use this name to access the column in data views, rules, and other areas within PolicyCenter.</p> <p>IMPORTANT All column names on a table must be unique within that table. Otherwise, Studio displays an error if you verify the resource and the application server fails to start.</p>	None
nullok	<p>Whether the column can contain null values.</p> <p>In general, this is always true, as many tables include columns that do not require a value at different points in the process.</p>	true
overwrittenInStagingTable	<p><i>Internal.</i> If true and the entity is loadable, the loader process auto-populates the staging table during import.</p> <p>IMPORTANT If set to true, do not attempt to populate the table yourself as the loader import process overwrites this table.</p>	false
scalable	Whether this value scales as the effective and expired dates change. This attribute applies only to number-type values. For example, you cannot scale a varchar. Also, it only applies to effective dated types.	false
setterScriptability	See “Data Objects and Scriptability” on page 158 for information.	all
soapnullok	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	None

<column> attribute	Description	Default
supportsLinguisticSearch	<p>Applies only to columns of varchar-based data types.</p> <ul style="list-style-type: none"> If true, searches performed on this field are linguistic. If false, searches are binary. <p>IMPORTANT You cannot use this attribute with an encrypted column. If you attempt to do so, PolicyCenter generates an error message upon resource verification.</p> <ul style="list-style-type: none"> 	false
type	<p><i>Required.</i> Data type of the column, or field. In the base configuration, Guidewire defines a number of data types and stores their metadata definition files (*.dti) in the following locations:</p> <ul style="list-style-type: none"> modules/p1/config/datatypes for system-level data types modules/pc/config/datatypes for optional application-specific data types <p>Each metadata definition <i>file name</i> is the name of a specific data type. You use one of these data types as the type attribute on the <column> element. Thus, the list of valid values for the type attribute is the same as the set of .dti files in the application datatypes folders.</p> <p>Each metadata definition also defines the <i>value type</i> for that data type. The value type determines how PolicyCenter treats that value in memory.</p> <p>The name of the data type is not necessarily the same as the name of its value type. For example, for the bit data type, the name of the data type is bit and the corresponding value type is java.lang.Boolean. Similarly, the data type varchar has a value type of java.lang.String.</p> <p>The datetime data type is a special case. PolicyCenter persists this data type in the application database using the <i>database</i> data type TIMESTAMP. This corresponds to the value type java.util.Date. In other words:</p> <ul style="list-style-type: none"> PolicyCenter represents a column whose type is datetime in memory as instances of java.util.Date. PolicyCenter stores this type of value in the database as TIMESTAMP. <p>WARNING Do not attempt to modify a base configuration data type file. You can invalidate your PolicyCenter application and prevent it from starting thereafter.</p> <p>See also</p> <ul style="list-style-type: none"> For general information data on types, see “Data Types” on page 233. For a list of the data types that you can modify or customize, see “Customizing Base Configuration Data Types” on page 237. For information on how to define new data types, see “Defining a New Data Type: Required Steps” on page 241. 	None

Subelements of <column>

The <column> element contains the following subelements:

<column> subelement	Description	Default
columnParam	See “<columnParam> Subelement” on page 184.	None
fulldescription	See “<fulldescription>” on page 194.	None
localization	See “<localization> Subelement” on page 186.	None

<columnParam> Subelement

You use the <columnParam> element to set parameters that a column type requires. The type attribute of a column determines which parameters you can set or modify by using the <columnParam> subelement. You can determine the list of parameters that a column type supports by looking up the type definition in its .dti file.

For example, if you have a `mediumtext` column, you can determine the valid parameters for that column by examining file `mediumtext.dti`. This file indicates that you can modify the following attributes of a `mediumtext` column:

- `encryption`
- `logicalSize`
- `trim whitespace`
- `validator`

Because you cannot modify the base configuration data type declaration files, you cannot see these files in Guidewire Studio. To view these files, navigate to the following directories:

- **System-level data types** – `modules/pl/config/datatypes`
- **Optional, application-specific data types** – `modules/pc/config/datatypes`

The following example, from `Account.eti` in PolicyCenter, illustrates how to use this subelement to define certain column parameters.

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
    desc="An account is ..."
    entity="Account"
    ...
    table="account"
    type="retireable">
    ...
    <column desc="Business and Operations Description."
        name="BusOpsDesc"
        type="varchar">
        <columnParam name="size" value="240"/>
    </column>
    ...
</extension>
```

Parameters that You Can Define by Using `<columnParam>`

The following list describes the parameters that you can define by using `<columnParam>`. These parameters are valid with many of data types, but not all of them.

Parameter	Description
<code>encryption</code>	Whether PolicyCenter stores this column in encrypted format. This only applies to text-based columns. Guidewire allows indexes on encrypted columns, or fields. However, because Guidewire stores encrypted fields as encrypted in the database, you must encrypt the input string and search for an exact match to it.
<code>logicalSize</code>	The size of this field in the PolicyCenter interface. You can use this value for String columns that do not have a maximum size in the database, such as CLOB objects. If you specify a value for the <code>size</code> parameter, then the <code>logicalSize</code> value must be less than or equal to the value of that parameter.
<code>precision</code>	The <i>precision</i> of the field. Precision is the total number of digits in the number. The <code>precision</code> parameter applies only if the data type of the field allows a precision attribute.
<code>scale</code>	The <i>scale</i> of the field. Scale is the number of digits to the right of the decimal point. The <code>scale</code> parameter applies only if the data type of the field allows a scale attribute.
<code>size</code>	Integer size value for columns of type TEXT and VARCHAR. Use with these column types <i>only</i> . This parameter specifies the maximum number of characters, not bytes, that the column can hold. WARNING The database upgrade utility automatically detects definitions that lengthen or shorten a column. For shortened columns, the utility assumes that the instigator of the change wrote a version check or otherwise verified that the change does not truncate existing column data. For both Oracle and SQL Server, if shortening a column causes the truncation of data, the ALTER TABLE statement in the database fails and the upgrade utility fails.
<code>trim whitespace</code>	Applies to text-based data types. If true, then PolicyCenter automatically removes leading and trailing white space from the data value.
<code>validator</code>	The name of a ValidatorDef in <code>fieldvalidators.xml</code> . See " <code><ValidatorDef></code> " on page 261.

The following parameters are specific to certain data types:

Parameter	Use with data type	Description
currencyProperty	currencyamount	Name of a property on the owning entity that returns the currency for this column.
secondaryAmountProperty	currencyamount	Name of a property on the owning entity that returns the secondary amount related to this currency amount column.
exchangeRateProperty	currencyamount	Name of a property on the owning entity that returns the exchange rate to use during currency conversions.
countryProperty	localizedstring	Name of a property on the owning entity that returns the country to use for localizing the data format for this column.

See also

- See “Overriding Data Type Attributes” on page 217 for an example of using a nested `<columnParam>` subelement within a `<column-override>` element to set the `encryption` attribute on a column.

`<localization>` Subelement

The `<localization>` subelement has one attribute, `tableName`, which is the table name of the localization join table. See “Localized Columns in Entities” on page 61 in the *Globalization Guide* for a discussion of the column `<localization>` element with examples on how to use it.

`<componentref>`

To review, a Component data object is similar to a compound property in that it represents a group of fields that all go together. A common example is a MoneyComponent that represents a monetary amount. This money component includes a numeric amount and the currency type for that monetary amount.

To reference a Component object from another data object, you use the ComponentRef object element. Guidewire defines this element in the data model metadata files as the `<componentRef>` XML subelement.

Attributes of `<componentref>`

The `<componentref>` element contains the following attributes:

<code><componentref></code> attribute	Description	Default
<code>deprecated</code>	If true, then PolicyCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference. If you deprecate an item, use the description to explain why. For more information, see “The deprecated Attribute” on page 178.	false
<code>desc</code>	A description of the purpose and use of the array.	None
<code>exportable</code>	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
<code>flatten</code>	Whether to flatten the <code><component></code> by exposing a single property that represents the <code><component></code> or inflate the <code><component></code> by exposing the individual fields of the component. For example, an entity includes a component called MoneyComponent. If the <code>flatten</code> attribute is true, the entity has a single property that returns the MoneyComponent. If the <code>flatten</code> attribute is false, the entity exposes the Amount and Currency properties of the MoneyComponent as if they were properties of the entity itself.	false
<code>generateCode</code>	<i>Internal.</i>	true
<code>getterScriptability</code>	See “Data Objects and Scriptability” on page 158 for information.	all

<componentref> attribute	Description	Default
name	<i>Required.</i> Specifies the name of the property on the entity. The value of this attribute is important only if the value of flatten is false.	None
prefix	An optional prefix to use if defining properties that include a component. You must use a prefix if you include the same component twice on the same entity.	None
ref	<i>Required.</i> The name of the component to include, such as MoneyComponent.	None
setterScriptability	See “Data Objects and Scriptability” on page 158 for information.	all

Subelements of <componentref>

The <componentref> element contains the following subelements:

<componentref> subelement	Attributes	Description	Default
annotation	• name • value	Creates a name and value pair. The subelement must contain both attributes.	None
fulldescription	None	See “<fulldescription>” on page 194.	

<edgeForeignKey>

You use the <edgeForeignKey> element to define a reference to another entity, in a manner similar to the <foreignkey> element. However, you use an edge foreign key in place of a standard foreign key to break a cycle of foreign keys in the data model. Guidewire defines this element in the data model metadata files as the <edgeForeignKey> XML subelement.

The Data Model and Circular References

A chain of foreign keys can form a cycle, also known as a *circular reference*, in the data model. As an example of a circular reference, entity type A has a foreign key to entity type B, and B has a foreign key to A. Circular references can occur with more extensive chains of foreign keys, such as A refers to B, which refers to C, which refers to A. The PolicyCenter data model does not permit circular foreign keys reference, because PolicyCenter cannot determine a safe order for committing the entity instances in a circular reference to the database.

For example, entity type A has a foreign key to entity type B have foreign key references to each other. The foreign keys create a circular reference. Suppose that a bundle contains a new instance of A and a new instance of B. The circular reference would cause a foreign key constraint to fail upon committing the bundle. If PolicyCenter commits A before B is committed and in the database, a constraint failure occurs on the foreign key from A to B. The converse order of committing B before A causes a similar failure.

An edge foreign key in place of a standard foreign key resolves circular references so PolicyCenter can determine a safe order for committing the entity instances within a cycle. An edge foreign key from A to B introduces a new, hidden associative entity with a foreign key to A and a foreign key to B. The edge foreign key associates A and B without establishing foreign keys in the database directly between them. With an edge foreign key, PolicyCenter can safely first commit new object A, then new object B, and finally the edge foreign key instance.

Edge Foreign Keys in Entity Database Tables

Unlike a standard foreign key, an edge foreign key does not correspond to an actual column on the database table of an entity type. Nor does an edge foreign key implement a database foreign key constraint. However, the PolicyCenter *Data Dictionary* labels edge foreign keys as standard foreign keys. In Gosu code, you access edge foreign keys in the same manner that you access standard foreign keys.

Edge Foreign Keys and Associative Database Tables

An edge foreign key creates an *associative table* in the database. An associative table is essentially a table of foreign keys relationships. An associative table associates other database tables with each other but holds no other essential business data itself.

In PolicyCenter, the associative table that implements an edge foreign key has two columns:

- OwnerID
- ForeignEntityID

If entity instance A has an edge foreign key to entity type B, PolicyCenter creates a row in the edge foreign key table. The value in the row for OwnerID points to A and the value for ForeignEntityID points to B.

Every time you traverse, or de-reference, the edge foreign key, PolicyCenter loads the join array.

- If the array is of size 0, then the value of the edgeForeignKey is null.
- If the array is of size 1, the PolicyCenter follows the ForeignEntityID on the row.

Edge Foreign Keys in Gosu

In Gosu code, edge foreign keys work in a manner similar to standard foreign keys. Just like a standard foreign key, you can query an edge foreign key and get and set its attributes.

Edge Foreign Keys and Performance

An edge foreign key has more performance issues than a standard foreign key, because PolicyCenter must manage a separate table for the relationship. Queries must join an extra table. Nullability constraints in the database do not work with edge foreign keys, so you must enforce nullability constraints with extra Gosu code the you develop.

Edge Foreign Keys and Archiving

Entity types that are part of the domain graph used to archive policies must implement the `Extractable` delegate. Their metadata definitions include the XML element `<implementsEntity name="Extractable"/>`. Otherwise, the server refuses to start.

If you add an edge foreign key to an entity that is part of the domain graph, the edge foreign key must also implement the `Extractable` delegate. Edge foreign keys do not inherit the `<implementsEntity>` delegate from their enclosing entities. If you do mark edge foreign keys in extractable entities as themselves extractable, the server refuses to start.

See also

- “The Archiving Domain Graph” on page 247
- “Delegate Data Objects” on page 161
- “`<implementsEntity>`” on page 194

When to Use Edge Foreign Keys

Use an edge foreign key only to avoid circular foreign key references in the data model. Circular foreign key references can prevent PolicyCenter from determining a safe order for committing the entity instances in a circular reference to the database.

Use an edge foreign key instead of standard foreign key in the following situations:

- An entity type has self-referencing foreign keys, including foreign keys between subtypes.
- A Group entity must specify its parent group.
- Entity type A has a foreign key to B, and entity type B has a foreign key to A.

- Cycles that involve more than two entity types.
- The primary member of an array requires a foreign key to its owner.

Attributes of <edgeForeignKey>

The <edgeForeignKey> element contains the following attributes.

<edgeForeignKey> attribute	Description	Default
createhistogram	Whether to create a histogram on the column during an update to the database statistics. Note: It is possible to override this attribute on an existing column in an extension (*.etx) file using the <column-override> element. You can use the override to turn off an existing histogram or to create one that did not previously exist. This change does not take effect during an upgrade. The change occurs <i>only</i> if you regenerate statistics for the affected table by using the Guidewire maintenance_tools command.	false
deprecated	See also <ul style="list-style-type: none"> “Working with Attribute Overrides” on page 216 “Configuring Database Statistics” on page 42 in the <i>System Administration Guide</i> “Maintenance Tools Command” on page 159 in the <i>System Administration Guide</i> If true, then PolicyCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference. If you deprecate an item, use the description to explain why. For more information, see “The deprecated Attribute” on page 178.	false
desc	A description of the purpose and use of the edge foreign key.	None
edgeTableName	The name of the edge table entity. If you do not specify one, then PolicyCenter creates one automatically.	None
edgeTableName	<i>Required.</i> The name of the edge, or join array, table to create.	None
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
exportasid	If specified, PolicyCenter exposes the field in SOAP APIs as a string, whose value represents the PublicID of the referenced object.	false
fkentity	<i>Required.</i> The entity to which this foreign key points.	None
generateCode	<i>Internal.</i>	true
getterScriptability	See “Data Objects and Scriptability” on page 158 for information.	all
ignoreforevents	If you change (or add, or remove) an entity X that does not generate events, then PolicyCenter searches for all event-generating entity instances that specify X. If PolicyCenter finds any of these event-generating entity instances, it generates Changed events for those entity instances. To determine what entities reference a non-event-generating entity, PolicyCenter examines the foreign keys and arrays that point to the entity. However, if you set ignoreForEvents to true on an entity that references the non-event-generating entity, then PolicyCenter ignores that link as it determines what entities specify another entity. <ul style="list-style-type: none"> At the entity level, the ignoreForEvents attribute means changes to (or addition or removal of) this entity do not cause Changed events to fire for any other entity. At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events. 	false

<edgeForeignKey> attribute	Description	Default
importableagainstexistingobject	If true and the entity is importable, or loadable, then the value in the staging table can be a reference to an existing object. This reference is the publicID of a row in the source table for the referenced object.	true
loadable	If true, then PolicyCenter creates a staging table for the edge table.	false
name	<i>Required.</i> Specifies the name of the property on the entity.	None
nullok	Whether the column can contain null values. This value is meaningless for edgeForeignKey objects.	true
overwrittenInStagingTable	<i>Internal.</i> If true and the edge table is loadable, the loader process auto-populates the staging table during import. IMPORTANT If set to true, do not attempt to populate the table yourself as the loader import process overwrites this table.	false
setterScriptability	See “Data Objects and Scriptability” on page 158 for information.	all
soapnullok	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	None

Subelements of <edgeForeignKey>

IMPORTANT The <edgeForeignKey> element does not inherit the <implementsEntity> delegate from its enclosing entity. You must specify a value for the name attribute on <implementsEntity> if you wish to associate a delegate with this edge foreign key.

<edgeForeignKey> subelement	Attributes	Description
fulldescription	None	See “<fulldescription>” on page 194.
implementsEntity	<ul style="list-style-type: none"> • adapter – Interrelated with the requires attribute on <delegate> • name – name of delegate entity to implement (required = true) 	<p>Applies to the edge table type created by the <edgeForeignKey>.</p> <p>For a description for the requires attribute, see “Delegate Data Objects” on page 161.</p>

<events>

If the <events> element appears within an entity, it indicates that the entity raises events. Usually, the code indicates the standard events (add, change, and remove) by default. If the <events> element does not appear in an entity, that entity does not raise any events. You cannot modify the set of the events associated with a base entity through extension. However, you can add additional events to a base entity through extension, even if that entity already contains a set of predefined events.

Note: This element is not valid for a nonPersistentEntity.

Guidewire defines this element in the data model metadata files as the <events> XML subelement. There can be at most one <events> element in an entity. However, you can specify additional events through the use of <event> subelements. For example:

```
<events>
  <event>
    ...
  </events>
```

Note: PolicyCenter automatically adds the EventAware delegate to any entity that contains the <events> element.

Attributes of <events>

There are no attributes on the <events> element.

Subelements of <events>

The <events> element contains the following subelements.

<events> subelement	Description
event	<p>Defines an additional event to fire for the entity. Use multiple <event> elements to specify multiple events. This subelement contains the following attributes:</p> <ul style="list-style-type: none"> • description (required = true) • name (required = true) <p>The attributes are self-explanatory. The <event> element requires each one.</p>

<foreignkey>

The <foreignkey> element defines a foreign key reference to another entity.

Attributes of <foreignkey>

The <foreignkey> element contains the following attributes.

<foreignkey> attribute	Description	Default
columnName	<p><i>Optional.</i> If specified, PolicyCenter uses this value as the column name of the corresponding database column. If you do not specify a columnName value, then PolicyCenter uses the value of the name attribute for the database column name.</p> <p>Note: As a common and recommended practice, use the suffix ID for the column name. For example, for a foreign key with name Policy, set the columnName to PolicyID.</p> <p>Guidewire does not require that you use an ID suffix on names of foreign key columns. However, Guidewire strongly recommends that you adopt this practice to help you analyze the database and identify foreign keys.</p> <p>IMPORTANT All column names on a table must be unique in that table. Otherwise, Studio displays an error if you verify the resource, and the application server fails to start.</p>	None
createConstraint	If true, the database creates a foreign key constraint for this foreign key.	true
createBackingIndex	If true, the database automatically creates a backing index on the foreign key. If set to false, the database does not create a backing index. See “Attribute createBackingIndex” on page 193 for more information.	true
createHistogram	Whether to create a histogram on the column during an update to the database statistics. Note: It is possible to override this attribute on an existing column in an extension (*.etx) file using the <column-override> element. You can use the override to turn off an existing histogram or to create one that did not previously exist. This change does not take effect during an upgrade. The change occurs only if you regenerate statistics for the affected table by using the Guidewire maintenance_tools command.	false
See also <ul style="list-style-type: none"> • “Working with Attribute Overrides” on page 216 • “Configuring Database Statistics” on page 42 in the <i>System Administration Guide</i> • “Maintenance Tools Command” on page 159 in the <i>System Administration Guide</i> 		

<foreignkey> attribute	Description	Default
deprecated	If true, then PolicyCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference. If you deprecate an item, use the description to explain why. For more information, see "The deprecated Attribute" on page 178.	false
desc	A description of the purpose and use of the field.	None
existingreferencesallowed	If the following attributes are set to false, which is not the default: <ul style="list-style-type: none">• loadable• importableagainstexistingobject then, the value in the staging table can only be a reference to an existing object.	true
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
exportasid	If specified, PolicyCenter exposes the field in SOAP APIs as a string, whose value represents the PublicID of the referenced object.	false
fkentity	<i>Required.</i> The entity to which this foreign key refers.	None
generateCode	<i>Internal.</i>	true
getterScriptability	See "Data Objects and Scriptability" on page 158 for information.	all
ignoreforevents	If you change (or add, or remove) an entity X that does not generate events, then PolicyCenter searches for all event-generating entity instances that specify X. If PolicyCenter finds any of these event-generating entity instances, it generates Changed events for those entity instances. To determine what entities reference a non-event-generating entity, PolicyCenter examines the foreign keys and arrays that point to the entity. However, if you set ignoreForEvents to true on an entity that references the non-event-generating entity, then PolicyCenter ignores that link as it determines what entities specify another entity. <ul style="list-style-type: none">• At the entity level, the ignoreForEvents attribute means changes to (or addition or removal of) this entity do not cause Changed events to fire for any other entity.• At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events.	false
importableagainstexistingobject	If true and the entity is importable (loadable), then the value in the staging table can be a reference to an existing object. (This is the publicID of a row in the source table for the referenced object.)	true
includeIdInIndex	If true, then include the ID as the last column in the backing index for the foreign key. This is useful if the access pattern in one or more important queries is to join to this table through the foreign key. You can then use the ID to probe into a referencing table. The only columns that you need to access from the table are this foreign key, and the retired and ID columns. In that case, adding the ID column to the index creates a covering index and eliminates the need to access the table.	false
loadable	If true, you can load the field through staging tables. A staging table can contain a column for the public ID of the referenced entity.	true
name	<i>Required.</i> Specifies the name of the property on the entity.	None

<foreignkey> attribute	Description	Default
nonEffDated	This applies only to foreign keys that are between effdated elements. If a foreign key is between effdated elements and this attribute is true, then PolicyCenter creates a real foreign key between the elements. Essentially, you use this attribute to specify a foreign key to a specific version of the given entity.	false
nullok	Whether the field can contain null values.	true
overwrittenInStagingTable	<i>Internal.</i> If true (and the table is loadable), it indicates that the loader process auto-populates the staging table during import. IMPORTANT If set to true, do not attempt to populate the table yourself because the loader import process overwrites this table.	false
owner	If true, it indicates that even if it is a foreign key, the row from the other table that this key references is a child node.	false
setterScriptability	See “Data Objects and Scriptability” on page 158 for information.	all
soapnullok	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	None
triggersValidation	Whether changes to the entity referred to by this foreign key trigger validation.	false

Attribute `createBackingIndex`

Suppose you want to create a unique index on a single, nullable foreign key column. You must turn off the automatic creation of a backing index on the foreign key. If the database automatically creates a backing index and you add a unique index, the database identifies the unique index as redundant and removes it.

The following example entity illustrates this concept.

```

<entity xmlns="http://guidewire.com/datamodel"
        desc="Table for testing unique indexes that allow nulls"
        entity="TestUniqueAllowsNulls"
        javaClass="com.guidewire.px.domain.test.TestUniqueAllowsNulls"
        platform="false"
        table="test_uniq Allows_nulls"
        type="retireable">

    <column desc="Importable column" name="A" type="integer"/>

    <foreignkey columnName="FKTestUniqueID"
                desc="Primary address associated with the contact.
                      User chose to not have a backing index for this foreign key."
                fkentity="TestUnique" name="TestUniqueID"
                owner="true"
                triggersValidation="true"
                createBackingIndex="false"/>

    <index desc="This index is unique but should allow nulls since the column is nullable
              and is not redundant"
           name="FKTestUniqueID"
           unique="true">
        <indexcol keyposition="1"
                  name="FKTestUniqueID"/>
    </index>

</entity>
```

Subelements of <foreignkey>

The <foreignkey> element contains the following subelements.

<foreignkey> subelement	Attributes	Description
fulldescription	None	See “<fulldescription>” on page 194.

<fulldescription>

PolicyCenter uses the `fulldescription` subelement to populate the *Data Dictionary*. For example:

```
<fulldescription>
  <![CDATA[<p>Aggregates the information needed to display one activity row
  (base entity for all other activity views).</p>]]>
</fulldescription>
```

<implementsEntity>

The `<implementsEntity>` subelement specifies that an entity implements the specified delegate. Guidewire calls an entity an *implementor* of a delegate if the entity specifies the delegate in a `<implementsEntity>` subelement.

IMPORTANT Do not change the delegate that a Guidewire base entity implements by creating an extension entity that includes an `<implementsEntity>` subelement. PolicyCenter generates an error if you do.

If a delegate definition includes the optional `requires` attribute, then the implementor must provide an `adapter` attribute on its `<implementsEntity>` subelement. The `adapter` attribute specifies the name of a Java or Gosu type that implements the interface that the delegate definition specifies in its own `requires` attribute.

For example, the PolicyCenter base configuration defines a `Cost` delegate as follows:

```
<?xml version="1.0"?>
<delegate ... name="Cost" requires="gw.api.domain.financials.CostAdapter">
  ...
</delegate>
```

The base configuration defines a `BACost` entity that includes an `<implementsEntity>` subelement, which specifies delegate with `name="Cost"`. Therefore, the `BACost` entity is an implementor of the `Cost` delegate.

```
<?xml version="1.0"?>
<entity ... entity="BACost" ... >
  ...
    <implementsEntity name="Cost" adapter="gw.lob.ba.financials.BACostAdapter" />
  ...
</entity>
```

The `Cost` delegate requires an implementation of the `CostAdapter` interface. So in its `adapter` attribute, the `BACost` entity specifies a `BACostAdapter` class, which implements the `CostAdapter` interface that the `Cost` adapter specifies in its `requires` attribute.

Follow these rules for defining entities that implement delegates:

- If you specify a value for the `requires` attribute in a delegate, then implementers of the delegate must specify an `adapter` attribute in their definitions. The `adapter` attribute must specify the name of a Java or Gosu type that implements the interface specified by the `requires` attribute in delegate definition.
- If you do not specify a value for the `requires` attribute in a delegate, then implementers of the delegate must not specify an `adapter` attribute their definitions.

The Extractable Delegate

Entities that are part of the domain graph must implement the `Extractable` delegate. If you add an edge foreign key to an entity that is part of the domain graph, then the edge foreign key must also implement the `Extractable` delegate.

For example, if you create a custom subtype of `Contact`, then the custom subtype must implement the `Extractable` delegate. Edge foreign keys do not inherit delegate definitions from their enclosing entity.

WARNING Entities that are part of the domain graph must implement the `Extractable` delegate by using the `<implementsEntity>` element. Otherwise, the server refuses to start.

Attributes of <implementsEntity>

The <implementsEntity> element contains the following attributes.

<implementsEntity> subelement	Description
adapter	The name of the type that implements the interface specified by the <code>requires</code> attribute on <delegate>. You must specify this value if you set a value for the <code>requires</code> attribute. Otherwise, do not provide a value.
name	<i>Required.</i> The name of the delegate that this entity must implement.

Subelements of <implementsEntity>

There are no subelements on the <implementsEntity> subelement.

<implementsInterface>

The <implementsInterface> subelement specifies that an entity implements the specified interface. This element defines two attributes, an interface (`iface`) attribute and an implementation (`impl`) attribute. The <implementsInterface> subelement requires both attributes.

For example, the PolicyCenter base configuration defines the BACost entity with the following <implementsInterface> subelement:

```
<entity ... entity="BACost" ...>
  ...
  <implementsInterface
    iface="gw.lob.ba.financials.BACostMethods"
    impl="gw.lob.ba.financials.BACostMethodsImpl"/>
</entity>
```

The BACostMethods interface has getter methods that any class which implements this interface must provide. The getter methods are coverage, state, and vehicle. By including the <implementsInterface> subelement, the BACost entity lets you use getter methods on instances of the BACost entity in Gosu code.

```
var cost : BACost
var cov     = cost.Coverage
var state   = cost.State
var vehicle = cost.Vehicle
```

Attributes of <implementsInterface>

The <implementsInterface> element contains the following attributes.

<implementsInterface> subelement	Description
iface	<i>Required.</i> The name of the interface that this data object must implement.
impl	<i>Required.</i> The name of the class or subclass that implements the specified interface.

Subelements on <implementsInterface>

There are no subelements on the <implementsInterface> subelement.

<index>

The <index> element defines an index on the database table used to store the data for an entity. Guidewire defines this element in the data model metadata files as the <index> XML subelement. This element contains a required subelement, which is <indexcol>.

The <index> element instructs PolicyCenter to create an index on the physical database table. This index is in addition to those indexes that PolicyCenter creates automatically.

An index improves the performance of a query search within the database. It consists of one or more fields that you can use together in a single search. You can define multiple `<index>` elements within an entity, with each one defining a separate index. If a field is already part of one index, you do not need to define a separate index containing only that field.

For example, PolicyCenter frequently searches non-retired accounts for one with a particular account number. Therefore, the `Account` entity defines an index containing both the `Retired` and `AccountNumber` fields. However, another common search uses just `AccountNumber`. Since that field is already part of another index, a separate index containing only `AccountNumber` is unnecessary.

You cannot use an `<index>` element with the `<nonPersistentEntity>` element.

IMPORTANT In general, the use of a database index has the possibility of reducing update performance. Guidewire recommends that you add a database index with caution. In particular, do not attempt to add an index on a column of type CLOB or BLOB. If you do so, PolicyCenter generates an error message upon resource verification.

Attributes of `<index>`

The `<index>` element contains the following attributes.

<code><index></code> attribute	Description	Default
<code>clustered</code>	<i>Unused.</i>	<code>false</code>
<code>desc</code>	A description of the purpose and use of the index.	<code>None</code>
<code>expectedtobecovering</code>	If <code>true</code> , it indicates that the index covers all the necessary columns for a table that is to be used for at least one operation, for example, search by name. Thus, if <code>true</code> , it indicates that there is to be no table lookup. In this case, use the <code>desc</code> attribute to indicate which operation that is.	<code>false</code>
<code>name</code>	<i>Required.</i> The name of the index. The first character of the name must be a letter. The maximum value for an index name is 18 characters. IMPORTANT For <code><subtype></code> definitions, all index names must be unique between the subtype and supertype. In other words, do not duplicate an index name between the subtype definition in the <code>extensions</code> folder and its supertype in the <code>metadata</code> folder. Otherwise, PolicyCenter generates an error on resource verification.	<code>None</code>
<code>trackUsage</code>	If <code>true</code> , track the usage of this index.	<code>true</code>
<code>unique</code>	Whether the values of the index are unique for each row.	<code>false</code>
<code>verifyInLoader</code>	If <code>true</code> , then PolicyCenter runs an integrity check for unique indexes before loading data from the staging tables.	<code>true</code>

Subelements of <index>

The <index> element contains the following subelements.

<index> subelement	Description	Default
forceindex	<p>Use to force PolicyCenter to create an index if running against a particular database.</p> <p>This subelement is useful because the index generation algorithm can throw away some declared indexes as being redundant. In some cases, PolicyCenter can require one or more of those indexes to work around an optimization problem.</p> <p>This subelement contains the following attributes:</p> <ul style="list-style-type: none"> • oracle – If true, force the creation of an index if running against an Oracle database. • sqlserver – If true, force the creation of an index if running against a Microsoft SQL Server database. 	None
indexcol	<p><i>Required.</i> Defines a field that is part of the index. You can specify multiple <indexcol> elements to define composite indexes. This subelement contains the following attributes:</p> <ul style="list-style-type: none"> • keyposition – <i>Required.</i> The position of the field within the index. The first position is 1. • name – <i>Required.</i> The column name of the field. This can be a column, foreignkey, or typekey defined in the entity. • sortascending – If true, which is the default, then the sort direction is ascending. 	None

<onetoone>

The <onetoone> element defines a single-valued association to another entity that has a one-to-one cardinality. Guidewire defines this element in the data model metadata files as the <onetoone> XML subelement. A one-to-one element functions in a similar manner to a foreign key in that it makes a reference to another entity. However, its purpose is to provide a reverse pointer to an entity or object that is pointing at the <onetoone> entity, through the use of a foreign key.

For example, entity A has a foreign key to entity B. You can associate an instance of B with at most one instance of A. Perhaps, there is a unique index on the foreign key column. This then defines a one-to-one relationship between A and B. You can then declare the <onetoone> element on B, to provide simple access to the associated A. In essence, using a one-to-one element creates an *array-of-one*, with, at most, one element. Zero elements are also possible.

Note: PolicyCenter labels one-to-one elements in the Guidewire *Data Dictionary* as foreign keys. You access these elements in Gosu code in the same manner as you access foreign keys.

Attributes of <onetoone>

The <onetoone> element contains the following attributes.

<onetoone> attribute	Description	Default
cascadeDelete	If true, then PolicyCenter deletes the entity to which the <onetoone> element points if you delete this entity.	false
deprecated	If true, then PolicyCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.	false
	<p>If you deprecate an item, use the description to explain why.</p> <p>For more information, see “The deprecated Attribute” on page 178.</p>	
desc	A description of the purpose and use of the field.	None

<onetoone> attribute	Description	Default
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
fkentity	<i>Required.</i> The entity to which this foreign key points.	None
generateCode	<i>Internal.</i>	true
getterScriptability	See "Data Objects and Scriptability" on page 158 for information.	all
ignoreforevents	If you change (or add, or remove) an entity X that does not generate events, then PolicyCenter searches for all event-generating entity instances that specify X. If PolicyCenter finds any of these event-generating entity instances, it generates Changed events for those entity instances. To determine what entities reference a non-event-generating entity, PolicyCenter examines the foreign keys and arrays that point to the entity. However, if you set ignoreForEvents to true on an entity that references the non-event-generating entity, then PolicyCenter ignores that link as it determines what entities specify another entity. <ul style="list-style-type: none"> • At the entity level, the ignoreForEvents attribute means changes to (or addition or removal of) this entity do not cause Changed events to fire for any other entity. • At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events. 	false
linkField	<i>Optional.</i> Specifies the foreign key field that points back to this object.	None
name	<i>Required.</i> Specifies the name property on the entity.	None
nullok	Whether the field can contain null values.	true
owner	If true, this entity owns the linked object (the object to which the <onetoone> element points): <ul style="list-style-type: none"> • If you delete the owning object, then PolicyCenter deletes the linked object as well. • If you update the object pointed to by the <onetoone> element, then PolicyCenter considers the owning object updated as well. 	false
setterScriptability	See "Data Objects and Scriptability" on page 158 for information.	all
triggersValidation	Whether changes to the entity pointed to by this entity trigger validation.	false

Subelements of <onetoone>

The <onetoone> element contains the following subelements.

<onetoone> subelement	Description	Default
fulldescription	See "<fulldescription>" on page 194.	None

<remove-index>

The <remove-index> element defines the name of a database index that you want to remove from the data model. It is valid for use with the following data model elements:

- <entity>
- <extension>

You can use this element to safely remove a non-Primary key index if it is one of the following:

- The index is non-unique.
- The index is unique but contains an ID column.

Guidewire performs metadata validation to ensure that the <remove-index> element removes only those indexes that fall into one of these categories.

The Index is Non-unique

You can safely remove a non-primary key index with the `unique` attribute set to `false`. In general, these are indexes that Guidewire provides for performance enhancement. It is safe to remove these kinds of indexes.

The Index is Unique, But Contains an ID Column

You can safely remove a non-Primary key index with the `unique` attribute set to `true` if that index includes ID as a key column. For example, the `WorkItem` entity contains the following index definition:

```
<index desc="Covering index to speed up checking-out of work items and they involve search on status"
       name="WorkItemIndex2" unique="true">
    <indexcol keyposition="1" name="status"/>
    <indexcol keyposition="2" name="Priority" sortascending="false"/>
    <indexcol keyposition="3" name="CreationTime"/>
    <indexcol keyposition="4" name="ID"/>
</index>
```

Even though the `unique` attribute is set to `true`, you can safely remove this index because the index definition contains an ID column, `keyposition="4"`. These types of indexes do not enforce a uniqueness condition. Thus, it is safe to remove these kinds of indexes.

Attributes of `<remove-index>`

The `<remove-index>` element contains the following attributes.

<code><remove-index></code> attribute	Description	Default
<code>name</code>	Name of the database index to remove.	None

Using the `<remove-index>` Element

In many cases, you simply want to modify an existing database index. In that case, use the `<remove-index>` element to remove the index, then simply add an index – with the same name – that contains the desired characteristics.

`<typekey>`

The `<typekey>` element defines a field for which a typelist defines the values. Guidewire defines this element in the data model metadata files as the `<typekey>` XML subelement.

Note: For information on typelists, typekeys, and keyfilters, see “Working with Typelists” on page 265.

Attributes of <typekey>

The <typekey> element contains the following attributes.

<typekey> attribute	Description	Default
columnName	<p><i>Optional.</i> If specified, PolicyCenter uses this value as the column name of the corresponding database column. If you do not specify a columnName value, then PolicyCenter uses the value of the name attribute for the database column name.</p> <p>IMPORTANT All column names on a table must be unique within that table. Otherwise, Studio displays an error if you verify the resource and the application server fails to start.</p>	None
createhistogram	<p>Whether to create a histogram on the column during an update to the database statistics.</p> <p>Note: It is possible to override this attribute on an existing column in an extension (*.etx) file using the <column-override> element. You can use the override to turn off an existing histogram or to create one that did not previously exist.</p> <p>This change does not take effect during an upgrade. The change occurs <i>only</i> if you regenerate statistics for the affected table by using the Guidewire maintenance_tools command.</p> <p>See also</p> <ul style="list-style-type: none"> • “Working with Attribute Overrides” on page 216 • “Configuring Database Statistics” on page 42 in the <i>System Administration Guide</i> • “Maintenance Tools Command” on page 159 in the <i>System Administration Guide</i> 	false
default	The default value given to the field during new entity creation.	None
deprecated	<p>If true, then PolicyCenter marks the typekey as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.</p> <p>If you deprecate a typekey, use the description attribute (desc) to explain why.</p> <p>For more information, see “The deprecated Attribute” on page 178.</p>	false
desc	A description of the purpose and use of the field.	None
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
generateCode	<i>Internal.</i>	true
getterScriptability	See “Data Objects and Scriptability” on page 158 for information.	None
loadable	If true, then you can load the field through staging tables. A staging table can contain a column, as a String, for the code of the typekey.	true
name	<i>Required.</i> Specifies the name of the property on the entity	None
nullok	Whether the field can contain null values.	true
overwrittenInStagingTable	<i>Internal.</i> If true and the typekey is loadable, the loader process auto-populates the typekey in the staging table during import.	false
	IMPORTANT If set to true, do not attempt to populate the typekey yourself because the loader import process overwrites this typekey.	
setterScriptability	See “Data Objects and Scriptability” on page 158 for information.	None
soapnullok	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	None

<typekey> attribute	Description	Default
typefilter	The name of a filter associated with the typelist. See "Static Filters" on page 276 for additional information.	None
typelist	<p><i>Required.</i> The name of the typelist from which this field gets its value.</p> <p>See also</p> <p>"Working with Typelists" on page 265.</p>	None

Subelements of <typekey>

The <typekey> element contains the following subelements.

<typekey> subelement	Description	Default
keyfilters	Defines one or more <keyfilter> elements. There can be at most one <keyfilters> element in an entity. See "Dynamic Filters" on page 280 for additional information.	None
fulldescription	See "<fulldescription>" on page 194.	None

Subelements of <keyfilters>

The <keyfilters> element contains the following subelements.

<keyfilters> subelement	Description	Default
<keyfilter>	<p>Specifies a keyfilter to use to filter the typelist. This element requires the <name> attribute.</p> <p>This attribute defines a relative path, navigable through Gosu dot notation, to a <i>physical</i> data field. Each element in the path must be a data model field.</p> <p>Note: You can include multiple <keyfilter> elements to specify multiple keyfilters.</p>	None

Working with Associative Arrays

This topic describes the different types of associative arrays that Guidewire provides as part of the base data model configuration.

This topic includes:

- “Overview of Associative Arrays” on page 203
- “Subtype Mapping Associative Arrays” on page 205
- “Typelist Mapping Associative Arrays” on page 207

Overview of Associative Arrays

In its simplest terms, an associative array provides a mapping between a set of *keys* and the *values* that the keys represent. A common example of this type of mapping is a telephone book, in which a name maps to a telephone number. Another common example is a dictionary, which maps terms to their definitions.

To expand on this concept, a telephone book contains a set of names, with each name a key and the associated telephone number the value. Using array-like notation, you can write:

```
telephonebook[peter] = 555-123-1234  
telephonebook[shelly] = 555-234-2345  
...
```

PolicyCenter uses associate arrays to expose array values as a typesafe map within Gosu code. The following example uses a typekey from a State typelist as the mapping index for an associative array of state capitals:

State typekey index	Maps to...
Capital[State.TC_AL]	Montgomery
Capital[State.TC_AK]	Juneau
Capital[State.TC_AZ]	Phoenix
Capital[State.TC_AR]	Little Rock

There are two necessary tasks in working with an associative array in Gosu:

- Exposing the key set to the type system
- Calculating the value from the key

Associative Array Mapping Types

An associative array must have a key that maps to a value. The mapping type describes what PolicyCenter uses as the key and what value that key returns.

Mapping type	Key	Value
Subtype mapping	Entity subtype	Implicit subtype field on an entity
TypeList mapping	TypeList	Typekey field on the entity

To implement an associative array, add one of the following elements to an `<array>` element in the data type definition file. The number of results that each returns—the cardinality of the result set—depends on the element type.

<code><link-association></code>	Returns at most one element. The return type is an object of the type of the array.
<code><array-association></code>	Returns an array of results that match the typekey. The number of results can be zero, one, or more.

Each `<array>` element in a data type definition file can have zero to one of each of these elements.

As an example, in the ClaimCenter `Claim` definition file (`configuration → config → Metadata → Entity → Claim.eti`), you see the following XML (simplified for clarity):

```

<entity xmlns="http://guidewire.com/datamodel"
        entity="Claim"
        table="claim"
        type="retireable">
    ...
    <array arrayentity="ClaimMetric"
           desc="Metrics related to this claim."
           exportable="false"
           ignoreforevents="true"
           name="ClaimMetrics"
           triggersValidation="false">
        <link-association>
            <subtype-map/>
        </link-association>
        <array-association>
            <typeList-map field="ClaimMetricCategory"/>
        </array-association>
    </array>
    ...
</entity>

```

See also

- For examples of how to create a subtype associative array, see “Subtype Mapping Associative Arrays” on page 205.
- For examples of how to create a typeList associative array, see “TypeList Mapping Associative Arrays” on page 207.

Scriptability and Associative Arrays

It is possible to set the following attributes on each `<link-association>` and `<array-association>` element:

- `hasGetter`
- `hasSetter`

For example:

```
<link-association hasGetter="true" hasSetter="true">
  <typelist-map field="TAccountType"/>
</link-association>
```

For these attributes:

- If `hasGetter` is `true`, then you can read the property.
- If `hasSetter` is `true`, then you can update the property.

Note: If you do not specify either of these attributes, then PolicyCenter defaults to `hasGetter="true"`.

See also

- “Data Objects and Scriptability” on page 158

Issues with Setting Array Member Values

There are several issues with setting associative array member values, including:

1. You can use a query builder expression to retrieve a specific entity instance. However, the result of the query is read-only. You must add the retrieved entity to a bundle to be able to manipulate its fields. To work with bundles, use one of the following:

```
var bundle = gw.transaction.Transaction.getCurrent()
gw.transaction.Transaction.runWithNewBundle(\ bundle -> ) //Use this version in the Gosu tester
```

2. You can only set array values on fields that are database-backed fields, not fields that are derived properties. To determine which fields are derived, consult the PolicyCenter *Data Dictionary*.

See also

- See “Overview of the Query Builder APIs” on page 127 in the *Gosu Reference Guide* for information on working with query builder expressions.

Subtype Mapping Associative Arrays

You use subtype mapping to access array elements based on their subtype. In other words, this type of associative array divides the elements of the array into multiple partitions, each of which contains only array elements of a particular object *subtype*. For example, in the ClaimCenter base configuration, the data model defines an associative array called `ClaimMetrics` on the `Claim` object.

In the `Claim` definition file (`configuration → config → Metadata → Entity → Claim.eti`), you see the following (simplified) XML:

```
<entity xmlns="http://guidewire.com/datamodel"
  entity="Claim"
  table="claim"
  type="retireable">
  ...
  <array arrayentity="ClaimMetric"
    desc="Metrics related to this claim."
    exportable="false"
    ignoreforevents="true"
    name="ClaimMetrics"
    triggersValidation="false">
    <link-association>
      <subtype-map/>
    </link-association>
  </array>
  ...
</entity>
```

The array—`ClaimMetrics`—contains a number of objects, each of which is a subtype of a `ClaimMetric` object. The data model defines the associative array using the `<link-association>` element. A link associations return

at most one element and the return type is an object of the type of the array. In this case, the return type is an object of type `ClaimMetric`, or more specifically, one of its subtypes.

The ClaimCenter data model defines a number of subtypes of the `ClaimMetric` object, including:

- `DecimalClaimMetric`
- `IntegerClaimMetric`
- `AllEscalatedActivitiesClaimMetric`
- `OpenEscalatedActivitiesClaimMetric`
- ...

To determine the complete list of subtypes on an object, consult the *Data Dictionary*. The dictionary organizes the subtypes into a table at the top of the dictionary page with active links to sections that describe each subtype in greater detail.

Working with Array Values Using Subtype Mapping

To retrieve an array value through subtype mapping, use the following syntax:

```
base-entity.subtype-map.property
```

Each field has the following meanings:

<code>base-entity</code>	The base object on which the associative array exists, for example, the <code>Claim</code> entity for the <code>ClaimMetrics</code> array.
<code>subtype-map</code>	The array entity subtype, for example, <code>AllEscalatedActivitiesClaimMetric</code> (a subtype of <code>ClaimMetric</code>).
<code>property</code>	A field or property on the array object. For example, the <code>AllEscalatedActivitiesClaimMetric</code> object contains the following properties (among others): <ul style="list-style-type: none"> • <code>ClaimMetricCategory</code> • <code>DisplayTargetValue</code> • <code>DisplayValue</code>

Note: To see a list of subtypes for any given object, consult the PolicyCenter *Data Dictionary*. To determine the list of fields (properties) on an object, again consult the *Data Dictionary*.

Example One

The following example code uses the sample data in the Guidewire ClaimCenter base configuration. It first retrieves a specific claim object using a query builder and then uses that object as the base entity from which to retrieve array member properties.

```
var clm = gw.api.database.Query.make(Claim).compare("ClaimNumber", Equals,
  "235-53-365870").select().getAtMostOneRow()

print("AllEscalatedActivitiesClaimMetric\tClaim Metric Category = "
  + clm.AllEscalatedActivitiesClaimMetric.ClaimMetricCategory.DisplayName)
print("AllEscalatedActivitiesClaimMetric\tDisplay Value = "
  + clm.AllEscalatedActivitiesClaimMetric.DisplayValue)
print("AllEscalatedActivitiesClaimMetric\tReach Yellow Time = "
  + clm.AllEscalatedActivitiesClaimMetric.ReachYellowTime)
```

The output of running this code in the Gosu tester looks similar to the following:

```
AllEscalatedActivitiesClaimMetric      Claim Metric Category = Claim Activity
AllEscalatedActivitiesClaimMetric      Display Value = 0
AllEscalatedActivitiesClaimMetric      Reach Yellow Time = null
```

Example Two

The following sample code:

- Retrieves a read-only claim object.
- Adds the claim object to transaction bundle to make it writable.

- Sets a specific property on the `AllEscalatedActivitiesClaimMetric` object (a subtype of the `ClaimMetric` object) associated with the claim.

If you recall from the definition of the `claim` object, `ClaimCenter` associates an array of `ClaimMetric` objects—the `ClaimMetrics` array—with the `Claim` object. The metadata definition file also defines the `ClaimMetrics` array as being of type `<link-association>` using subtypes. Thus, you can access array member properties by first accessing the array member of the proper subtype.

```
uses gw.transaction.Transaction

var todaysDate = java.util.Date.getCurrentDate
var clm = gw.api.database.Query.make(Claim).compare("ClaimNumber", Equals,
    "235-53-365870").select().getAtMostOneRow()

//Query result is read-only, need to get current bundle and add object to bundle
var bundle = Transaction.getCurrent()
clm = bundle.add(clm)

print("AllEscalatedActivitiesClaimMetric\tReach Yellow Time = "
    + clm.AllEscalatedActivitiesClaimMetric.ReachYellowTime)
clm.AllEscalatedActivitiesClaimMetric.ReachYellowTime = todaysDate

print("\nAfter modifying the ReachYellowTime value...\n")
print("AllEscalatedActivitiesClaimMetric\tReach Yellow Time = "
    + clm.AllEscalatedActivitiesClaimMetric.ReachYellowTime)
```

The output of running this code in the `Gosu` tester looks similar to the following:

```
AllEscalatedActivitiesClaimMetricReach Yellow Time = null

After modifying the ReachYellowTime value...

AllEscalatedActivitiesClaimMetricReach Yellow Time = 2010-05-21
```

For more information making query results writable, see “Adding Entity Instances to Bundles” on page 339 in the *Gosu Reference Guide*.

TypeList Mapping Associative Arrays

You use a typelist map to partition array objects based on a typelist field (typecode) in the `<array>` element. In the `ClaimCenter` base configuration, the `ClaimMetrics` array on `Claim` contains a typelist mapping and the previously described subtype mapping.

```
<entity xmlns="http://guidewire.com/datamodel"
    entity="Claim"
    table="claim"
    type="retireable">
...
<array arrayentity="ClaimMetric"
    desc="Metrics related to this claim."
    exportable="false"
    ignoreforevents="true"
    name="ClaimMetrics"
    triggersValidation="false">
    <array-association>
        ...
        <typelist-map field="ClaimMetricCategory"/>
    </array-association>
</array>
...
</entity>
```

The `<typelist-map>` element requires that you set a value for the `field` attribute. This attribute specifies the typelist to use to partition the array.

IMPORTANT It is an error to specify a typelist mapping on a field that is not a typekey.

Associative arrays of type <array-association> are different from those created using <link-association> in that they can return more than a single element. In this case, the code creates an array of `ClaimMetric` objects named `ClaimMetrics`. Each `ClaimMetric` object, and all subtype objects of it, contain a property called `ClaimMetricCategory`. The array definition code utilizes that fact and uses the `ClaimMetricCategory` typelist as a partitioning agent.

The `ClaimMetricCategory` typelist contains three typecodes, which are:

- `ClaimActivityMetrics`
- `ClaimFinancialMetrics`
- `OverallClaimMetrics`

Each typecode specifies a category, which contains multiple `ClaimMetric` object subtypes. For example, the `OverallClaimMetrics` category contains two `ClaimMetric` subtypes:

- `DaysInitialContactWithInsuredClaimMetric`
- `DaysOpenClaimMetric`

In another example from the ClaimCenter base configuration, you see the following defined for `ReserveLine`.

```
<entity entity="ReserveLine"
  xmlns="http://guidewire.com/datamodel"
  ...
  table="reserveline"
  type="retireable">
  ...
  <array arrayentity="TAccount"
    arrayfield="ReserveLine"
    name="TAccounts"
    ...
    <link-association hasGetter="true" hasSetter="true">
      <typelist-map field="TAccountType"/>
    </link-association>
  </array>
  ...
</entity>
```

In this case, the array definition code creates a <link-association> array of `TAccount` objects and partitions the array by the `TAccountType` typelist typecodes.

Working with Array Values Using Typelist Mapping

To retrieve an array value through typelist mapping, use the following syntax:

```
entity.typecode.property
```

Each field has the following meaning:

<code>entity</code>	The object on which the associative array exists, for example, the <code>ReserveLine</code> entity on which the <code>Taccounts</code> array exists
<code>typecode</code>	The typelist typecode that delimits this array partition, for example, <code>OverallClaimMetrics</code> (a typecode from the <code>ClaimMetricCategory</code> typelist).
<code>property</code>	A field or property on the array object. For example, the <code>ClaimMetric</code> object contains the following properties (among others): f <ul style="list-style-type: none"> • <code>ReachRedTime</code> • <code>ReachYellowTime</code> • <code>Skipped</code>

Example One

The following example code uses the sample data in the Guidewire ClaimCenter base configuration. It iterates over the members of the `ClaimMetrics` array that fall into the `OverallClaimMetrics` category. (The `ClaimMetricCategory` typelist contains multiple type codes, of which `OverallClaimMetrics` is one.)

```
uses gw.api.database.Query

var clm = Query.make(Claim).compare("ClaimNumber", Equals, "235-53-365870").select().FirstResult
for (time in clm.OverallClaimMetrics) {
```

```

        print(time.Subtype.DisplayName + ": ReachYellowTime = " + time.ReachYellowTime)
    }

```

The output of running this code in the Gosu tester looks something similar to the following:

```

Initial Contact with Insured (Days): ReachYellowTime = 2010-09-27
Days Open: ReachYellowTime = 2011-04-08

```

Example Two

The following example code also uses the sample data in the Guidewire ClaimCenter base configuration. It first retrieves a specific `Claim` object and then retrieves a specific `ReserveLine` object associated with that claim.

```

var clm = gw.api.database.Query.make(Claim).compare("ClaimNumber", Equals,
    "235-53-365870").select().FirstResult
var thisReserveLine = clm.ReserveLines.first()

print(thisReserveLine)
print(thisReserveLine.cashout.CreateTime)

```

The output of running this code in the Gosu tester looks something similar to the following:

```

(1) 1st Party Vehicle - Ray Newton; Claim Cost/Auto body
Fri Oct 08 16:14:50 PDT 2010

```

Setting Array Member Values

The following example code also uses the sample data in the Guidewire ClaimCenter base configuration. It uses a query builder expression to retrieve a specific claim entity. As the result of the query is read-only, you must first retrieve the current bundle, then add the claim to the bundle to make its fields writable. The retrieved claim is the base entity on which the `ClaimMetrics` array exists.

The following sample code:

- Retrieves a read-only claim object.
- Adds the claim object to transaction bundle to make it writable.
- Sets specific properties on the `ClaimMetric` object associated with the claims that are in the `OverallClaimMetrics` category.

If you recall from the definition of the claim object, ClaimCenter associates an array of `ClaimMetric` objects—the `ClaimMetrics` array—with the `Claim` object. The metadata definition file also defines the `ClaimMetrics` array as being of type `<array-association>` using the `ClaimMetricCategory` typelist. Thus, you can access array member properties by first accessing the array member of the proper category.

```

uses gw.transaction.Transaction
uses gw.api.database.Query

var todaysDate = java.util.Date.getCurrentDate
var thisClaim = Query.make(Claim).compare("ClaimNumber", Equals, "235-53-365870").select().FirstResult
//Query result is read-only, need to get current bundle and add entity to bundle

var bundle = Transaction.getCurrent()
thisClaim = bundle.add(thisClaim)

//Print out the current values for the ClaimMetric.ReachYellowTime field on each subtype
for (color in thisClaim.OverallClaimMetrics) {
    print("Subtype - " + color.Subtype.DisplayName + ": ReachYellowColor = " + color.ReachYellowTime)
}

print("\nAfter modifying the values...\n")

//Modify the ClaimMetric.ReachYellowColor value and print out the new values
for (color in thisClaim.OverallClaimMetrics) {
    color.ReachYellowTime = todaysDate
    print("Subtype - " + color.Subtype.DisplayName + ": ReachYellowColor = " + color.ReachYellowTime)
}

```

The output of running this code in the Gosu tester looks similar to the following:

```

Subtype - Initial Contact with Insured (Days): ReachYellowColor = 2010-10-13
Subtype - Days Open: ReachYellowColor = 2010-10-13

After modifying the values...

```

```
Subtype - Initial Contact with Insured (Days): ReachYellowColor = 2010-10-13
Subtype - Days Open: ReachYellowColor = 2010-10-13
```

For more information making query results writable, see “Adding Entity Instances to Bundles” on page 339 in the *Gosu Reference Guide*.

Modifying the Base Data Model

This topic discusses how to extend the base data model as well as how to create new data objects.

This topic includes:

- “Planning Changes to the Base Data Model” on page 211
- “Defining a New Data Entity” on page 214
- “Extending a Base Configuration Entity” on page 215
- “Working with Attribute Overrides” on page 216
- “Extending the Base Data Model: Examples” on page 218
- “Removing Objects from the Base Configuration Data Model” on page 227
- “Deploying Data Model Changes to the Application Server” on page 231

Planning Changes to the Base Data Model

Before proceeding to modify the base data model, Guidewire strongly recommends that you first review the *Data Dictionary*. Verify that the existing data model does not provide the functionality that you need first before modifying the base application functionality.

Overview of Data Model Extension

Entity extensions are additions to the entities in the base data model. Although you cannot modify the base data type declaration files directly, you can define an extension to one in a separate .etx file. You can also define new data model objects that extend the data model in an .eti file. This allows new PolicyCenter releases to modify the base definitions without affecting your extensions, thus preserving an upgrade path.

By extending the base data model, you can:

- Add fields (columns) to an existing base entity through the use of the <column>, <typekey>, <foreignkey>, <array>, and similar elements. See “Data Object Subelements” on page 177.
- Create a new entity with custom fields using any of the entity types listed in “Base PolicyCenter Data Objects” on page 160.

- Modify a small subset of the attributes of an existing base entity using overrides.
- Remove (or hide) an extension to a base entity that exists in the **extensions** folder as an **.etx** declaration file.
- Remove (or hide) a base entity that exists in the **extensions** folder as an **.eti** declaration file.

However, using extensions, you cannot:

- Delete a base entity or any of its fields. If you do not use a particular base entity or one of its fields, then simply ignore it.
- Change most of the attributes of a base entity or any of its fields.

Strategies for Extending the Base Data Model

Extending the data model means one of the following:

- You want to add new fields to an existing entity.
- You want to create a new entity.

During planning for data model extensions, you need to consider performance implications. For example, if you add hundreds of extensions to a major object, this can conceivably exceed a reasonable row size in the database.

Adding Fields to an Entity

If an entity has almost all the functionality you need to support your business case, you can add one or more fields to it. In this sense, Guidewire uses the term *field* to denote one of the following:

- Column
- Typekey
- Array
- Foreign key

See “Data Column and Field Types” on page 146 for a description of these fields.

Subtyping a Non-Final Entity

If you want to find a new use for an existing entity, you can subtype and rename it. For instance, suppose that you want to track individuals who have already had the role of **IssueOwner**. In this case, it can be useful to create **PastIssueOwner**.

Creating a New Entity

Occasionally, careful review of the base application data model makes it clear that you need to create a new entity. There are many types of base entities within Guidewire applications. However, Guidewire **strongly** recommends in general practice that you always use one of the following types if you create a new entity:

retireable	This type of entity is an extension of the editable entity. It is not possible to delete this entity. It is possible to retire it, however.
versionable	This type of entity has a version and an ID. It is possible to delete entities of this type from the database.

As a general rule, Guidewire recommends the following:

- Make the new entity **versionable** if it is not necessary for another entity to refer to the entity through the use of a foreign key.
- Make the entity **retirable** otherwise.

See “Data Entities and the Application Database” on page 155 for more information on these data types.

In general, you typically want to create a new entity under the following circumstances:

- If your business model requires an object that does not logically exist in the application. Or, if you have added too many fields to an existing entity, and want to abstract away some of it into a new, logical entity.
- If you need to manage arrays of objects, as opposed to multiple objects, you can create an entity array.

Reference Entities

To store some unchanging reference data, such as a lookup table that seldom changes, you can create a reference entity. An example of a business case for a reference entity is a list of typical reserve amounts for a given exposure. To avoid the overhead of maintaining foreign keys, make reference entities keyable. Unless you want to build in the ability to edit this information from within the application, set `setterscriptability = hidden`. This prevents Gosu code from accidentally overwriting the data.

Guidewire recommends that you determine that this is not really a case for creating a typelist before you create a reference entity. See “Defining a Reference Entity” on page 224 for more information.

What Happens If You Change the Data Model?

During server start up, PolicyCenter analyzes the metadata for changes since the last build. If you have made extensions, the application merges this into the working PolicyCenter data model which is the composite of the base entities and your extensions.

After merging the base data model with any extensions, PolicyCenter compares the startup layout to the physical schema in the current database. (Each PolicyCenter database stores schema version numbers and metadata checksums to optimize the analysis and comparison.)

If the application detects changes between the startup layout and the physical database schema, it initiates a database upgrade automatically. This keeps the physical schema synchronized with the schema defined by the XML metadata. By default, PolicyCenter refuses to start until the two are synchronized. By setting the `autoupgrade` parameter to `false` (within the `database` element in `config.xml`), you can configure PolicyCenter to report the need for an upgrade, but not actually perform it.

The PolicyCenter application server conducts a number of additional tests on the product model data model as it starts. See the following for more information:

- “Checking Product Model Availability” on page 105 in the *Product Model Guide*
- “Preventing Illegal Product Model Changes” on page 111 in the *Product Model Guide*
- “Verifying the Product Model” on page 117 in the *Product Model Guide*

WARNING Do not directly modify the physical database that PolicyCenter uses. Only make changes to the PolicyCenter data model through Guidewire Studio.

Database Upgrade Triggers

The upgrade utility initiates a database upgrade automatically at application server startup if there are additions, modifications, or extensions to any of the following:

- Data model version
- Extensions version
- Platform version
- PolicyCenter data model
- Field encryption
- Typelists

In addition to these generic changes, the following specific localization changes trigger a database upgrade:

- In file `localization.xml`, any change to the `<LinguisticSearchCollation>` subelement on the `<GWLocale>` element of the default application locale forces a database upgrade at application server startup.

- In file `collations.xml`, any change to the source definition of the DBJavaClass definition forces a database upgrade at application server startup.

Naming Restrictions for Extensions

PolicyCenter uses the names of extensions as the basis for several other internally-generated structures, such as database elements and Java classes. Because of this, it is important that you adhere to the naming requirements and guidelines described in this section.

IMPORTANT Deviations from these guidelines can result in product errors or unexpected behavior.

An extension name cannot start with a number; it must start with a letter. Other than that, an extension name can contain letters, numbers, or underscores (`_`). Guidewire does not permit any other characters in extension names.

Defining a New Data Entity

You define all new data entity objects in declaration files that end with the `.eti` extension. You do this through Guidewire Studio. Studio automatically manages the process and stores the `.eti` file in the correct location in the application (in the `Data Model Extensions → extensions` folder).

To create a new entity

1. Create a file for that entity through Studio:

- a. Navigate to `configuration → config → Extensions → Entity`.
- b. Right-click `Entity`, and then click `New → Entity`.

2. In the `Entity` dialog, specify the entity definition values.

Add fields to your new data entity. In the `Field` drop-down list, select the field type to add, and then click `Add +`. For example:

XML tag	Use to add
<code><array></code>	An array of entities
<code><column></code>	A field with a simple data type
<code><foreignkey></code>	A field referencing another entity
<code><typekey></code>	A field with a typelist

See “Data Object Subelements” on page 177 for information on the possible XML elements that you can add to your new entity definition.

3. Deploy your changes to the application server. You must redeploy the application after you make any change to the Guidewire PolicyCenter data model. See “Deploying Data Model Changes to the Application Server” on page 231 for details.

Extending a Base Configuration Entity

You define all of your entity-type extensions in files that end with the .etx extension. You do this through Guidewire Studio. Studio automatically manages the process and stores the .etx file in the correct location in the application.

IMPORTANT Guidewire provides certain entity extensions as part of the base application configuration. Many of the extension index definitions address performance issues. Other extensions provide the ability to configure the data model in ways that would not be possible if the extension was part of the base data model. Do not simply overwrite a Guidewire extension with your own extension without understanding the full implications of the change.

PolicyCenter extensions allow you to add new fields to the base data entities. You can add custom fields to extendable entities only. Not all entities are extendable, but most of the important business entities such as Policy, User, Contact, and others are extendable. (You can determine if an entity is extendable by looking in the *Data Dictionary* to see if it supports the `Extendable` attribute. The *Data Dictionary* displays the list of attributes for that entity type directly underneath the entity name.)

Use the `<extension>` XML root element to create an extension entity. Before creating a new extension file, first determine if one already exists.

- If an extension file for the entity does, then edit that file to extend the entity.
- If an extension file for the entity does not exist, then create the new extension file and populate it accordingly.

Do **not** attempt to create multiple extension files for the same entity. You can reference a given existing entity in only **one** extension (.etx or .tx) file. If you attempt to extend (or define) the same entity in multiple files, then the PolicyCenter application server generates an error at application start up. In all cases, Studio refuses to create entity or extension files with the same duplicate name.

To create a new extension file

The simplest (and safest) way to create a new extension file is to let Studio manage the process.

1. In the Studio Project window, navigate to `configuration` → `config` → `Metadata` → `Entity`, and then locate the entity that you want to extend.
2. Right-click the entity, and then click `New` → `Entity Extension`. Studio creates a basically empty extension file named `<entity>.etx`, places it in the `configuration` → `config` → `Extensions` → `Entity` folder, and opens it in a view tab for editing.

Note: If an extension file for the selected entity file already exists, Studio does not permit you to create another one. If the file name in the Entity Extension dialog box is grayed out, that means that an extension already exists. In that case, search in the `configuration` → `config` → `Extensions` → `Entity` folder for an existing extension file.

3. Populate the extension with the required attributes.
4. Deploy your changes to the application server. You must redeploy the application after you make any change to the Guidewire PolicyCenter data model. See “Deploying Data Model Changes to the Application Server” on page 231 for details.

Working with Attribute Overrides

It is possible to override certain attribute values (fields) on entities that Guidewire defines in files to which you do not have direct access. For example, you do not have write access to any entity definition files in the `configuration → config → Metadata → Entity` subfolders. Guidewire provides a limited number of override elements for use in `.etx` extension files in the `configuration → config → Extentions` folder.

To use an override element:

- If an entity extension file (`.etx`) already exists in the **Extensions** folder, then add one of the specified override elements to the existing file.
- If an entity extension file (`.etx`) does not already exist in the **Extensions** folder, then you need to create one and add an override element to that file.
- If an entity definition file (`.eti`) exists in the **Extensions** folder, then you can modify the original field definition. You do not need to use an override element.

Only add override elements to `.etx` files in the `configuration → config → Extentions` folder. Do not attempt to add an override element to a file in any other folder or to any other file type.

The following list describes the attributes that you can override by using an override element in an `.etx` file in the **Extensions** folder:

Override element	Attributes that you can override
<code><array-override></code>	<code>triggersValidation</code>
<code><column-override></code>	<code>createhistogram</code> <code>default</code> <code>nullok</code> <code>size</code> <code>supportsLinguisticSearch</code> <code>type</code>
<code><foreignkey-override></code>	<code>nullok</code> <code>triggersValidation</code>
<code><onetoone-override></code>	<code>triggersValidation</code>
<code><typekey-override></code>	<code>default</code> <code>nullok</code>

These attributes have the following meanings:

Attribute	Description
<code>createhistogram</code>	Use to turn on (or off) the creation of a histogram during the generation of table statistics. This change does not take effect during an upgrade. It only occurs if you regenerate statistics for the affected table using the Guidewire <code>maintenance_tools</code> command. For more information on the <code>createhistogram</code> attribute on the <code>column</code> element, see “ <code><column></code> ” on page 181.
<code>default</code>	Use to change the default value given to the field (column) during new entity creation.
<code>nullok</code>	Use to make the <code>nullok</code> attribute to be more restrictive. You can only make it more restrictive, for example, changing it from <code>nullok="true"</code> to <code>nullok="false"</code> .
<code>size</code>	Use to change the size of a column. See “A size Attribute Example” on page 217.
<code>supportsLinguisticSearch</code>	Use to enable linguistic search on a column.
<code>triggersValidation</code>	Use to determine if PolicyCenter initiates validation on changes to an array, a foreign key, or a one-to-one entity.
<code>type</code>	Use to change the data type of a column to a data type that is of a different value type. For example, suppose that you have a <code>String</code> column that currently is of <code>shorttext</code> and you want to make it use <code>longtext</code> . In this case, you use a <code><column-override></code> subelement to modify the original column definition.

Overriding Data Type Attributes

Besides the attributes that you can specifically override using the `<column-override>` element, you can also modify data type attributes on a column. You do this through the use of nested `<columnParam>` subelements within the `<column-override>` element.

For example, the base configuration Contact entity defines a TaxID column (in `Contact.eti`):

```
<column createhistogram="true"
       desc="Tax ID for the contact (SSN or EIN)."
       name="TaxID"
       type="ssn"/>
```

To encrypt the contents of this column (a reasonable course of action), create a Contact extension (`Contact.etx`) and use the `<column-override>` element to set the `encryption` attribute on the column:

```
<column-override name="TaxID">
  <columnParam name="encryption" value="true"/>
</column-override>
```

See also

- See “`<columnParam>` Subelement” on page 184 for a description of the `<columnParam>` element and the column attributes that you can modify using this element.

A size Attribute Example

You can change the size of the Name column for a Document entity as follows:

1. Open Guidewire Studio.
2. Navigate to **configuration** → **config** → **Metadata** → **Entity**, right-click `Document.eti`, and then click **Create Extension File**.
3. Before the final `</extension>` tag, insert the following code to set the size of the Name column to 100:

```
<column-override name="Name">
  <columnParam name="size" value="100"/>
</column-override>
```
4. Save the file.

A triggersValidation Example

You use the `triggersValidation` attribute to instruct PolicyCenter whether changes to an array, a foreign key, or a one-to-one entity initiates validation on that entity. To illustrate, in the base configuration, Guidewire defines the Account entity in file `Account.eti`.

```
<entity ... entity="Account" ...>
  ...
  <array arrayentity="UserRoleAssignment"
        desc="Role Assignments for this account."
        exportable="false"
        name="RoleAssignments"
        triggersValidation="true"/>
  ...
</entity>
```

The definition of the `RoleAssignments` array specifies that if any element of the array changes, the change triggers a validation of the object graph that includes the array. Suppose, for some reason, that you want to turn off validation even if changes occur to the `RoleAssignments` array. To do so, you need to create an extension file with an `<array-override>` element that modifies the `triggersValidation` attribute set on the base data object.

The following steps illustrate this concept.

To override a triggersValidation attribute

1. Create an `Account.etx` file.
 - a. Find the `Account.eti` file in the Studio configuration tree. You can use **Ctrl+N** to find the file.

- b.** Select the file, right-click and click **New → Entity Extension**.

Studio creates an `Account.etc` file and places it in the `configuration → config → Extentions → Entity` folder.

- 2.** Populate `Account.etc` with the following:

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="Account">
  <arrayOverride name="RoleAssignments" triggersValidation="false">
  </arrayOverride>
```

- 3.** Stop and restart the application server. The application server recognizes that there are changes to the data model and automatically runs the upgrade utility on start up.

This effectively switches off the validation that usually occurs on changes to elements of the `RoleAssignments` array.

Extending the Base Data Model: Examples

As described in “Defining a New Data Entity” on page 214, you can define entirely new custom entities that become part of the PolicyCenter entity model. You can then use these entities in your data views, rules, and Gosu classes in exactly the same way as you use the base entities. PolicyCenter makes no distinction between the usage of base entities and custom entities.

This topic describes the following:

- Creating a New Delegate Object
- Extending a Delegate Object
- Defining a Subtype
- Defining a Reference Entity
- Defining an Entity Array
- Extending an Existing View Entity

Testing Your Work

After you make any change to the data model, Guidewire recommends that you do the following to test your work.

- First, stop and restart Guidewire Studio. Verify that there are no errors or warnings. If there are, do not proceed until you have corrected the issues. Guidewire does not strictly require that you always stop and restart Studio after a data model change. However, it is one way to test that you have not inadvertently made a typing error, for example.
- After starting Studio, start Guidewire PolicyCenter. As the application server starts, it recognizes that you have made changes to the database and runs the upgrade utility automatically. Verify that the application server starts cleanly, without errors or warnings.

Creating a New Delegate Object

Creating a delegate object and associating it to an entity is a relatively straightforward process. It does involve multiple steps, as do many changes to the data model. To create a new delegate, you need to do the following:

Task	Description
Step 1: Create the Delegate Object	Define the delegate entity using the <code><delegate></code> element.
Step 2: Define the Delegate Functionality	Create a Gosu enhancement to provide any functionality that you want to expose on your delegate.

Task	Description
Step 3: Add the Delegate to the Parent Entity	Use the <implementsEntity> element to associate the delegate with the parent entity.
Step 4: Deploy your Data Model Changes	Deploy your data model changes. You may need to regenerate any Java API file or web service WSDL files after data model changes.

The following topics describe this process.

Step 1: Create the Delegate Object

The first step in defining a new delegate is to create the delegate file and populate it with the necessary code.

To create a delegate object

1. Within Guidewire Studio, navigate to **configuration** → **config** → **Extentions** → **Entity**.
2. Right-click and click **New** → **Entity Extension**.
3. Enter the file name, using the name of the delegate and adding the **.eti** extension. This action creates an empty file. You use this file to define the fields on the delegate.
4. Enter the delegate definition in the delegate file. If necessary, find an existing delegate file and use it as a model for the syntax.

For example, in the base configuration, Guidewire defines the implementation of the **Assignable** delegate as follows:

```
<delegate ... name="Assignable">
  ...
  <column desc="Time when entity last assigned"
    exportable="false"
    name="AssignmentDate"
    setterScriptability="hidden"
    type="datetime"/>
  <column desc="Date and time when this entity was closed. (Not applicable to all assignable entities)"
    exportable="false"
    name="CloseDate"
    type="datetime"/>
  <foreignkey columnName="AssignedGroupID"
    desc="Group to which this entity is assigned; null if none assigned"
    exportable="false"
    fkentity="Group"
    name="AssignedGroup"
    setterScriptability="ui"/>
  ...
</delegate>
```

Step 2: Define the Delegate Functionality

Next, you need to provide functionality for the delegate. While there are several ways to do this, you must use a Gosu enhancement implementation.

Java class implementation	In the base configuration, Guidewire provides a Java class implementation for each delegate to provide the necessary functionality. The Delegate object designates the Java class through the javaClass attribute. It is not possible for you to create and use a Java class for this purpose.
Gosu enhancement implementation	You must implement the delegate functionality through a Gosu enhancement that defines any functionality associated with the fields on the delegate. By providing the name of the delegate entity to the enhancement as you create it, you inform Studio that you are adding functionality for that particular delegate. Studio automatically recognizes that you are enhancing the delegate.

Step 3: Add the Delegate to the Parent Entity

The next step is to associate a delegate with an entity using the `<implementsEntity>` element in the entity definition.

- If you are creating a *new* entity, then you need to add the `<implementsEntity>` element to the entity definition .eti file.
- If you are working with an *existing* entity, then you need to add the `<implementsEntity>` element to the entity extension .etx file.

The following steps illustrate this process by creating a new entity. The steps to extend an existing entity are similar.

To associate a delegate with a new entity

1. Within Guidewire Studio, navigate to **configuration** → **config** → **Extentions** → **Entity**.
2. Right-click and select **New** → **Entity Extension** from the submenu.
3. Enter the name of the entity file. You must add the .eti extension. Studio does not do this for you. This action creates an empty file. You use this file to associate the delegate with your entity. If necessary, find an existing entity file and use it as a model for the syntax.

Note: Guidewire recommends that you add either the Ext prefix or suffix to all entities that you create or extend. If you do so, do so consistently. Always use prefixes or always use suffixes.

4. Enter the necessary text in this file, using the `<implementsEntity>` element to specify the delegate. For example (in the ClaimCenter base configuration), Guidewire defines the `Claim` entity—in `Claim.eti`—so that it implements a number of delegates, including the `Assignable` and `Validatable` delegates. The definition looks like this:

```
<entity xmlns="http://guidewire.com/datamodel" ... entity="Claim" ... />
<implementsEntity name="Validatable"/>
<implementsEntity name="Assignable"/>
...
...
```

Step 4: Deploy your Data Model Changes

After completing these steps, you need to deploy your data model changes. If necessary, see “Deploying Data Model Changes to the Application Server” on page 231 for details. Depending on whether you are working in a development or production environment, you need to perform different tasks. You may need to regenerate any Java API file or web service WSDL files after data model changes.

Extending a Delegate Object

Note: A Delegate data object is a reusable entity that contains an interface and a default implementation of that interface. See “Delegate Data Objects” on page 161 for more information.

Typically, you extend existing delegate objects to provide additional fields and behaviors on the delegate. Through extension, you can add the following to a delegate object in Guidewire PolicyCenter:

- `<column>`
- `<foreignkey>`
- `<description>`
- `<implementsEntity>`
- `<implementsInterface>`
- `<index>`
- `<typekey>`

You cannot remove base delegate fields. However, you can modify them to a certain extent—for example, by making an optional field non-nullable (but not the reverse). You cannot replace the `requires` attribute on the base delegate (which specifies the required adapter), but you can implement other delegates.

In Guidewire PolicyCenter, you can extend the following base configuration delegates:

- **Auditable**
- **Cost**
- **Coverable**
- **Coverage**
- **Exclusion**
- **Modifiable**
- **Modifier**
- **PCAssignable**
- **PolicyCondition**
- **RateFactor**
- **Transaction**
- **UWIssueDelegate**

Note: In addition to these application-specific delegates, you can extend the following system delegate:
`AddressAutofillable`.

You can only extend a delegate if the base configuration definition file for that delegate contains the following:
`extendable="true"`

The default for the `extendable` attribute on `<delegate>` is `false`. Therefore, if it is not set explicitly to `true` in the delegate definition file, you cannot extend that delegate.

Do not attempt to change the graph to which a Guidewire base entity belongs through extension. In other words, do not attempt to change the delegate that a Guidewire base entity implements through an extension entity using `<implementsEntity>`. PolicyCenter generates an error if you attempt to do so.

To extend a delegate object

1. Navigate to **configuration** → **config** → **Extentions** → **Entity**.
2. Right-click and select **New** → **Entity Extension**.
3. Enter the name of the delegate that you want to extend and add the `.etx` extension. Studio opens an empty file.
4. Enter the delegate definition in the delegate extension file. If necessary, find an existing delegate file and use it as a model for the syntax. For details, see “Creating a New Delegate Object” on page 218.

Modifier Delegate Example

To illustrate, in the base PolicyCenter configuration, Guidewire provides a **Modifier** delegate. PolicyCenter stores the delegate definition in file `Modifier.eti`, in **configuration** → **config** → **Metadata** → **Entity**. The following is a simplified version of this definition file.

```
<?xml version="1.0"?>
<delegate xmlns="http://guidewire.com/datamodel"
    effdatedOnly="true"
    extendable="true"
    javaClass="com.guidewire.pc.domain.policy.Modifier"
    name="Modifier"
    requires="gw.api.domain.ModifierAdapter">
    <fulldescription><![CDATA[A list of states and their rating factors (e.g. experience modification
        for workers' compensation).]]></fulldescription>
    <column desc="Boolean modifier"
        name="BooleanModifier"
        type="bit"/>
    ...
</delegate>
```

This delegate defines the following fields:

- **BooleanModifier**
- **DateModifier**
- **Eligible**

- Justification
- PatternCode
- RateModifier
- ReferenceDateInternal
- TypeKeyModifier
- ValueFinal
- State (type list)

Any entity that wants to use the **Modifier** delegate functionality must implement this delegate.

The Modifier Delegate Extension

Suppose that you want to extend the definition of the **Modifier** entity in file **Modifier.etx** to add additional columns to the delegate entity. For example:

- **OverriddenRateModifier**
- **SuggestedRateModifier**

The delegate extension code looks similar to the following.

```
<extension
    xmlns="http://guidewire.com/datamodel"
    entityName="Modifier">
    <column
        desc="The rate modifier that has been overridden."
        name="OverriddenRateModifier"
        type="rate">
    </column>
    <column
        desc="The rate modifier that the external system has suggested."
        name="SuggestedRateModifier"
        type="rate">
    </column>
</extension>
```

These fields are then accessible to any entity that implements this delegate. You use these fields to support user overrides of the modifier rates originally suggested by an external process. All modifiers for all lines of business use this override process. Thus, by placing this functionality in the **Modifier** delegate, each entity that implements the **Modifier** delegate inherits this functionality, instead of each entity implementing this functionality separately.

Each line of business (LOB) uses one or two LOB-specific modifier entities that implement the **Modifier** delegate. For example:

- **BAModifier**
- **BOPModifier**
- **CPModifier**
- **GLModifier**
- **PAModifier**
- **PAVehicleModifier**
- **ProductModifier**
- **WCModifier**

As each of these LOB-specific modifier entities already implements the **Modifier** entity, each automatically inherits the **Modifier** column extensions.

If an entity does not already implement the **Modifier** delegate, then you need to do one of the following:

- If an extension for that entity already exists in the **Extensions** folder, then you need to modify that extension file so that it implements the required delegate. In this case, modify the extension entity and add the following, using the appropriate adapter name:

```
<implementsEntity adapter="xxx" name="Modifier"/>
```

- If an extension for that entity does not exist in the **Extensions** folder, then you need to create an extension and have that entity extension implement the required delegate. In this case, right-click the extensions folder and select **New → Other file**. Enter the entity name and add the **.etx** extension. Populate the extension file with something similar to the following, using the appropriate adapter and entity name (**entityName**):

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="xxx">
  <implementsEntity adapter="xxx" name="Modifier"/>
</extension>
```

Modifier Delegates and Arrays

Delegate entities do **not** support arrays. Thus, the **Modifier** delegate does not include an array of **RateFactor** entities. Instead each concrete modifier type must define its own array of rate factors. Therefore, for each entity that implements the **Modifier** delegate (for example, **BAModifier**), there must also be a corresponding entity that implements the **RateFactor** delegate (for example, **BARateFactor**)

Modifier Delegates and Adapters

Each entity that implements the **Modifier** delegate must specify an adapter class. Each adapter class must implement the **gw.api.domain.ModifierAdapter** interface. This interface implements generic modifier operations such as returning the owning **Modifiable** entity and working with the array of rate factors.

Similarly, entities that implement the **RateFactor** delegate must separately specify an adapter class that implements the **gw.api.domain.RateFactorAdapter** interface. This adapter needs to define one method only, one that returns the owning **Modifier**.

To support out-of-sequence and preemption handling, each modifier type must implement its own matcher class. Each modifier and rate factor entity must separately implement the **gw.api.effdate.MatchableEffDated** interface. Guidewire provides the following classes as starting points for configuration:

- **gw.api.effdate.matcher.AbstractModifierMatcher.gs**
- **gw.api.effdate.matcher.AbstractRateFactorMatcher.gs**

See also

- “The PolicyCenter Data Model” on page 149
- “Delegate Data Objects” on page 161
- “**<implementsEntity>**” on page 194
- “Creating a New Delegate Object” on page 218
- “Quote Modifiers” on page 51 in the *Product Model Guide*

Defining a Subtype

A subtype is an entity that you base on another entity (its supertype). The subtype has all of the fields and elements of its supertype, and it can also have additional ones. You can also create subtypes of subtypes, with no limit to the depth of the hierarchy.

PolicyCenter does not associate a unique database table with a subtype. Instead, the application stores all subtypes in the table of its supertype. The supertype table includes a **subtype** column. The **subtype** column stores the **type** values for each subtype. PolicyCenter uses this column to resolve a subtype.

You define a subtype using the **<subtype>** element. You must specify certain attributes of the subtype, such as its name and its supertype (the entity on which PolicyCenter bases the subtype entity). For a description of required and optional attributes, see “Subtype Data Objects” on page 172.

Within the **<subtype>** definition, you must define its fields and other elements. For a description of the elements you can include, see “Data Object Subelements” on page 177.

Example

This example defines an **Inspector** entity as a subtype of **Person**. The **Inspector** entity includes a field for the inspector's license. To create the **InspectorExt.eti** file, navigate to the **Extensions** folder, then select **New → Entity Extension** from the right-click submenu. Enter the full name including the extension in the dialog.

```
<?xml version="1.0"?>
<subtype xmlns="http://guidewire.com/datamodel" desc="Professional inspector" displayName="Inspector"
    entity="InspectorExt"
    supertype="Person">
    <column name="InspectorLicenseExt" type="varchar" desc="Inspector's business license number">
        <columnParam name="size" value="30"/>
    </column>
</subtype>
```

Notice that while **InspectorExt** is subtype of **Person**, **Person**, itself, is a subtype of **Contact**. PolicyCenter automatically adds the new **InspectorExt** type to the **Contact** typelist. This is true, even though PolicyCenter marks the **Contact** typelist as **final**.

To see this change:

- To see this change in the *PolicyCenter Data Dictionary*, you must restart the application server.
- To see this change in the **Contact** typelist in Studio, you must restart Studio.

Defining a Reference Entity

You use a reference entity to store reference data for later access from within PolicyCenter without having to call out to an external application. For example, you can use reference entities to store:

- Medical payment procedure codes, descriptions, and allowed amounts
- Average reserve amounts, based on coverage and loss type
- PIP aggregate limits, based on state and coverage type

You can populate a reference entity by importing its data, and then you can query it using Gosu expressions. If you do not want PolicyCenter to update the reference data, set **setterScriptability = hidden** during entity definition.

IMPORTANT You can use any entity type as a reference entity. However, if you use the entity solely for storing and querying reference data, then Guidewire recommends that you use a **keyable** entity.

Example

This example defines a read-only reference table named **ExampleReferenceEntityExt**.

```
<entity entity="ExampleReferenceEntityExt" table="exampleref" type="keyable"
    setterScriptability="hidden">
    <column name="StringColumn" type="shorttext"/>
    <column name="IntegerColumn" type="integer"/>
    <column name="BooleanColumn" type="bit"/>
    <column name="TextColumn" type="longtext"/>
    <index name="internal1">
        <indexcol name="StringColumn" keyposition="1"/>
        <indexcol name="IntegerColumn" keyposition="2"/>
    </index>
</entity>
```

Defining an Entity Array

It is often useful to have a field that contains an array of other entities. For example, to represent that a contact can contain multiple address, the **Contact** entity contains the **Contact.ContactAddresses** field, which is an array of **ContactAddresses** entities for each **Contact** data object.

As you define the entity for the array, consider the type of entity to use. The general rule, again, is that if another entity does not refer to the new entity through a foreign key, then make the entity **versionable**. Otherwise, make the entity **retireable**.

To define an array of entities

1. Define the entity to use as a member of the array. Although you can use one of the PolicyCenter base entities for an array, it is often likely that you need to define a new entity for this purpose.
2. Define an array field in the entity that contains the array. You can give the field any name you want. It does not need to be the same name as the array entity.
3. Define a foreign key in the array entity that references the containing entity. PolicyCenter uses this field to connect an array to a particular data object.

For more information about	See
entity types	"Overview of Data Entities" on page 151
defining a new entity	"Defining a New Data Entity" on page 214
defining an array field	"<array>" on page 179
defining a foreign key field	"<foreignkey>" on page 191

Example

The following example, defines a new retireable entity named `ExampleRetireableArrayEntityExt` and adds it as an array to the `Policy` entity.

The first step is to define the array entity:

```
<?xml version="1.0"?>
<entity entity="ExampleRetireableArrayEntityExt" table="exampleretarray" type="retireable"
    exportable="true">
    <column name="StringColumn" type="shorttext"/>
    <typekey name="TypekeyColumn" typelist="SystemPermissionType" desc="A test typekey column"/>
    <foreignkey name="RetireableFKID" fkentity="ExampleRetireableEntityExt"
        desc="FK back to ExampleRetireableEntity" exportable="false"/>
    <foreignkey name="KeyableFKID" fkentity="ExampleKeyableEntityExt"
        desc="FK through to ExampleKeyableEntity" exportable="false"/>
    <foreignkey name="ClaimID" fkentity="Claim" desc="FK back to Claim" exportable="false"/>
    <implementsEntity name="Extractable"/>
    <index name="internal1" unique="true">
        <indexcol name="RetireableFKID" keyposition="1"/>
        <indexcol name="TypekeyColumn" keyposition="2"/>
    </index>
</entity>
```

To make this example useful, suppose that you now add this array field to the `Policy` entity. It is possible that a `Policy` entity already exists in the base configuration. Verify that the data type declaration file does not exist before adding another one. To determine if a `Policy` extension file already exists, use CTRL-N to search for `Policy.etx`.

- If the file does exist, then you can modify it.
- If the file does not exist, then you need to create one.

Add the following to `Policy.etx`.

```
<extension entityName="Policy" ...>
    ...
    <array arrayentity="ExampleRetireableArrayEntityExt"
        desc="An array of ExampleRetireableArrayEntityExt objects."
        name="RetireableArrayExt" />
    ...
</extension>
```

Next, modify the array entity definition so it includes a foreign key that refers to `Policy`:

```
<entity entity="ExampleRetireableArrayEntityExt" table="exampleretarray" ... >
    ...
    <foreignkey name="PolicyFK" fkentity="Policy"
        desc="FK from ExampleRetireableArrayEntityExt to Policy" />
```

```
<foreignkey name="PolicyID" fkentity="Policy" desc="FK back to Policy" exportable="false"/>
</entity>
```

Finally, create the two referenced entities, ExampleRetireableEntityExt and ExampleKeyableEntityExt.

Implementing a Many-to-Many Relationship Between Entity Types

To add a many-to-many relationship between entity types to the data model, you need to do the following:

- First, create a separate **versionable** entity.
- Add non-nullable foreign keys to each end of the many-to-many relationship.
- Add a unique index on each of the foreign keys.

These steps create a classic join entity.

The following example illustrates how to create a many-to-many relationship between **Account** and **Contact** entity types.

- It first creates a **versionable** entity type called **MyJoin**.
- It then defines foreign keys to **Account** and **Contact**.
- Finally, it adds indexes to these foreign keys.

The code looks similar to the following:

```
<entity xmlns="http://guidewire.com/datamodel"
    entity="MyJoin"
    table="myjoin"
    type="versionable"
    desc="Join entity modeling many-to-many relationship between Account and Contact entities">
    <foreignkey columnName="AccountID"
        fkentity="Account"
        name="Account"
        nullok="false"/>
    <foreignkey columnName="ContactID"
        fkentity="Contact"
        name="Contact"
        nullok="false"/>
    <index name="accountcontacts" unique="true">
        <indexcol keyposition="1" name="AccountID"/>
        <indexcol keyposition="2" name="ContactID"/>
    </index>
</entity>
```

To access the relationship, you need to add an array to one or both ends of the relationship. For example:

```
<extension xmlns="http://guidewire.com/datamodel" entityName="Account">
    <array arrayentity="MyJoin"
        desc="All the MyJoin entities related to Account."
        name="AccountContacts"/>
</extension>
```

This provides an array of **MyJoin** entities on **Account**.

Extending an Existing View Entity

Guidewire uses **viewEntity** entities to improve performance for list view pages in rendering the PolicyCenter interface. (See “**viewEntity** Data Objects” on page 174 for details.) Some default PCF pages make use of list view entities and some do not. If you add a new field to an entity, then you need to decide if you want to extend a **viewEntity** to include this new field. This can potentially avoid performance degradation.

The following example illustrates a case in which you add an extension both to a primary entity and its corresponding **viewEntity**. First search for **Activity.etc** to determine if one exists. (Use Ctrl+N to open the search dialog.)

Add the following to **Activity.etc**:

```
<extension entityName="Activity">
    ...
    <column type="bit"
        name="validExt"
```

```

    default="true"
    nullok="true"
    desc="Sample bit extension, with a default value."/>
...
</extension>

```

Next, search for `ActivityDesktopView.etx`. Suppose that you do not find this file, but you see that `ActivityDesktopView.eti` exists. As this is part of the base configuration, you cannot modify this declaration file. However, find the highlighted file in Studio and select **New → Entity Extension** from the right-click submenu. This opens a mostly blank file.

Enter the following in `ActivityDesktopView.etx`:

```

<viewEntityExtension entityName="ActivityDesktopView">
    <viewEntityColumn name="validExt" path="validExt"/>
</viewEntityExtension>

```

Note: The `path` attribute is always relative to the primary entity on which you base the view.

These data model changes add a `validExt` column (field) to the `Activity` object, which is also accessible from the `ActivityDesktopView` entity.

Extending an Existing View Entity with a Currency Column

If you create or extend a view entity that references a column that is of `type="currencyamount"`, you must handle the view entity extension in a particular manner. If the view entity extension references a `currencyamount` column, then you also need to define the `currencyProperty` that the original column definition specifies on the view entity as well.

Suppose, for example, that in ClaimCenter, you extend the `PriorClaimView` view entity by adding an `OpenReserves` field that has a path reference to `ClaimRpt.OpenReserves`:

```
<viewEntityColumn name="OpenReserves" path="ClaimRpt.OpenReserves"/>
```

Looking at the definition of `ClaimRpt`, you see the following:

```

<column default="0" desc="The open reserves." name="OpenReserves" nullok="false" type="currencyamount">
    <columnParam name="currencyProperty" value="ClaimCurrency"/>
</column>

```

Notice that `ClaimRpt.OpenReserves` is of `type="currencyamount"`. Thus, if you extend `PriorClaimView` with this field as indicated, you also need to add the following to `PriorClaimView.etx`:

```
<viewEntityTypekey name="ClaimCurrency" path="Currency"/>
```

By defining a `ClaimCurrency` column on the view entity, the view entity loads the claim currency as a part of the view entity. This makes the claim currency available to the `currrencyamount` column and ClaimCenter avoids loading the whole entity while dereferencing `Claim.Currency`.

Removing Objects from the Base Configuration Data Model

It is possible to safely remove certain objects from the base configuration data model. You can do this only if the data object declaration file exists in the `Data Model Extensions → extensions` folder, either as an `.eti` file or an `.etx` file.

The following table lists the objects that you can remove (or hide) in the base configuration:

Object to remove	Location	File	See
Base configuration <i>entity</i>	extensions	.eti	" Removing a Base Extension Entity " on page 228
Base configuration <i>extension</i>	extensions	.etx	" Removing an Extension to a Base Object " on page 229

Guidewire recommends that you review the material in “Implications of Modifying the Data Model” on page 229 before you remove an object from the data model.

IMPORTANT Guidewire provides certain entity extensions as part of the base application configuration. Many of the extension index definitions address performance issues. Other extensions provide the ability to configure the data model in ways that would not be possible if the extension was part of the base data model. Do not modify a Guidewire extension without understanding the full implications of the change.

WARNING Do not attempt to remove a base configuration data object (meaning one defined in the **Data Model Extensions → metadata** folder). Also, do not attempt to remove any extension marked as internal. Any attempt to do so can invalidate your Guidewire installation, causing the application server to refuse to start.

Removing a Base Extension Entity

It is possible to remove an extension entity that is part of the base data model. You can only remove an extension *entity* that the base configuration defines in the **configuration → config → Extentions → Entity** folder as an .eti file.

For example, in PolicyCenter, the base configuration includes a number of *entity extension* files in the **Extensions** folder, including:

- RateGLClassCodeExt
- RateWCClassCodeExt
- ...

There are two ways to remove an extension entity from the **Extensions** folder:

- For .eti files that Guidewire added as part of the base configuration, you need to edit the file, remove the current content, and insert a <deleteEntity> element in its place.
- For .eti files that you added as part of your customization process, you need merely delete the file.

In actual practice, you are not removing or deleting either the physical file or the extension itself. You are merely hiding—or negating—the effects of the extension entity in the data model.

See “Delete Entity Data Objects” on page 164 for information on the <deleteEntity> element.

To delete a base extension entity

1. Open the entity extension .eti file. This file must be located in the **extensions** folder.
 - If the .eti file is one that you created (meaning it is not part of the Guidewire-provided base configuration), then you merely need to delete the file. You can then omit the next step and continue to step 3.
 - If the .eti file is part of the Guidewire-provided base configuration, then continue to the next step.
2. Use the <deleteEntity> object to define the extension entity to remove from the data model. For example, if you want to remove an extension entity named RateGLClassCodeExt, then enter the following:

```
<?xml version="1.0"?>
<deleteEntity xmlns="http://guidewire.com/datamodel" name="RateGLClassCodeExt" />
```
3. Stop and restart the application server. At start up, the application server recognizes a data model change and automatically upgrades the database.

If you encounter error messages, or the application server refuses to start, examine your code and correct any issues before you attempt to continue.

Removing an Extension to a Base Object

It is also possible to remove an extension to a base data model object. You can only remove an entity *extension* that the base configuration defines in the **configuration → config → Extensions → Entity** folder as an .etx file.

As with the case with extension entities in the **Extensions** folder, there are two ways to handle the removal of entity extensions:

- For .etx files that Guidewire added as part of the base configuration, you need to edit the file, remove the current content, and insert a <deleteEntity> element in its place.
- For .etx files that you added as part of your customization process, you need merely delete the file.

IMPORTANT You cannot delete an extension marked as internal. Any attempt to do so can invalidate your Guidewire installation, causing the application server to refuse to start.

To remove a base extension

1. Navigate to the **extensions** folder and open the declaration file for the entity extension that you want to remove.
 - If the .etx file is one that you created (meaning it is not part of the Guidewire-provided base configuration), then you merely need to delete the file. You can then omit the next step and continue to step 3.
 - If the .etx file is part of the Guidewire-provided base configuration, then continue to the next step. For example, suppose that you want to remove (hide) the extension defined in the base configuration for the **Contact** entity. In that case, you open **Contact.etx** in the **Extensions** folder.
2. Delete the contents of the declaration file and insert a blank skeleton definition. For example, for the **Contact** extension, use the following:

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="Contact"/>
```
3. Stop and restart the application server. At start up, the application server recognizes a data model change and automatically upgrades the database.

If you encounter error messages, or the application server refuses to start, examine your code and correct any problems before you attempt to continue.

Implications of Modifying the Data Model

Any change to a data object modifies the underlying PolicyCenter database. Typically, each data entity has a corresponding table in the database and each object attribute maps to a table column. If you remove or alter a data object, the possibility exists that your object contains data such as rows in an entity table or data in a column.

This topic covers the following:

- Does Removing an Extension Make Sense?
- Writing SQL for Extension Removal
- Strategies for Handling Extension Removal
- Troubleshooting Modifications to the Data Model

Does Removing an Extension Make Sense?

Typically, removing a data object only makes sense in your development environment. If you build a new configuration, it can sometimes be necessary to remove an object rather than to drop it and to recreate the database. Dropping the database destroys any data that currently exists. This might not be an option if you share a database instance with multiple developers. In this case, removing the object is less painful for the development team.

During server start up, PolicyCenter checks for configuration changes, such as modified extensions, that require a database upgrade. Until the database reflects the underlying configuration, PolicyCenter refuses to start. If you have configured it to `autoupgrade` (in `config.xml`), the application upgrades the database on start up to match your modifications.

However, there are situations in which you modify a data object and the application upgrade process cannot make the corresponding database modification for you. Currently, the database upgrade tool is unable to implement extension modifications that require it to do any of the following:

- Change a column from nullable to non-nullable if `null` values exist in the database column or if there is not a default value. PolicyCenter refuses to start if there are `null` values in a non-nullable column.
- Change the underlying data type of a column, for example, changing a `varchar` column to `clob` or `varchar` column to `int`.
- Shorten the length of a `varchar/text-based` column (for example, `mediumtext` to `shorttext`) if this truncates data in the column. If shortening the length does not require truncating existing data, the upgrader can handle both shortening the length of a `varchar` column and increasing the length of a `varchar` column. (It can increase the length up to 8000 characters for SQL Server.)

Writing SQL for Extension Removal

Some modifications to the data model can require that you write an SQL statement to synchronize the database with the data model. How complex this SQL depends on what you want to remove. For example, to remove a field on an object, you need to alter the table and drop the column. However, if your extension includes foreign keys or indexes, then you need to take into account the referential integrity rules for the database—and your SQL becomes correspondingly more complex.

In a development environment, you can use the trial-and-error approach to writing your SQL.

In a production environment, in which—typically—there is data to preserve in each extension, the SQL can require an additional layer of complexity. For example, if you write an SQL statement in which a column type changes, your SQL can do something similar to the following:

- The SQL creates a temporary column.
- It copies data from the existing extension column to the temporary column.
- It drops the existing extension column.
- It recreates the extension column with its new properties as appropriate.
- It copies the data from the temporary column to the newly recreated column.
- It removes the temporary column.

However, in most cases, this is not necessary as Guidewire provides version triggers that modify the database automatically if the application detects data model changes. You only need to do manual SQL modification of the database if you want to modify your own extensions. Even in that case, Guidewire strongly recommends that a database administrator (DBA) always develop the SQL to use in removing an extension.

WARNING Be very careful of making changes to the data model on a live production database. You can invalidate your installation.

Strategies for Handling Extension Removal

Suppose that you have a development environment with multiple developers all using the same database instance. Before modifying the data model, first you need to communicate with your team to make them aware of what you plan to do. A good way to communicate your intentions is to provide the team with the SQL you intend to execute along with a list of impacted references files. After communicating with your team, follow a process similar to the following if removing a data object:

1. Remove the extension entity or entity extension using the methods outlined in the following sections:

- “Removing a Base Extension Entity” on page 228
 - “Removing an Extension to a Base Object” on page 229
2. Remove any references to the object in other parts of your configuration. If you do not remove these references, PolicyCenter displays error messages during server start-up.
3. Check in your changes.
4. Open an SQL command line appropriate to your server. For example, if you use Microsoft SQL Server, then open a query through the SQL Enterprise Manager.
5. Run your SQL statement to remove your extension.
6. Regenerate the toolkit.

In a production environment, Guidewire recommends that you include formal testing and quality assurance before removing or modifying an extension. Also, involve your company database administrator (DBA) and any impacted departments. Guidewire recommends also that you document your change and the reasons for it.

Troubleshooting Modifications to the Data Model

It is possible to change an `integer` column to a `typekey` column (and the reverse). However, `integer` values in the database do not necessarily map to a valid ID within the referenced typelist table after you make this type of change. Related to this, removing typecodes from a typelist (instead of retiring them) can cause data inconsistencies as well. If you have data that references a non-existent typecode, the upgrade does not complete and the server refuses to start. Instead of removing typecodes, retire them instead.

You can remove an extension field or the entire entity from the data model. If you do this, the server logs an informational message to the console such as:

```
pcx_ex_ProviderServicedStates: mismatch in number of columns - 5 in data model, 6 in physical database
```

Deploying Data Model Changes to the Application Server

How your deploy changes to the data model depends on if you are working in a *development* or *production* environment.

Development Environment

If you are working in a development environment, then do the following:

1. Use the following command (from the application `bin` directory) to regenerate the *Data Dictionary* so that it reflects your data model changes:

```
gwpc regen-dictionary
```

2. Stop and restart both the application server and Studio. As the application server and Studio share the same file structure in the development environment, you need only restart the development application server to pick up these changes.

If necessary (and it is almost always necessary if you change the data model), PolicyCenter runs the database upgrade tool during application start up.

Production Environment

If you are working in a production environment, then do the following:

1. Use the following command (from the application `bin` directory) to regenerate the *Data Dictionary* so that it reflects your data model changes:

```
gwpc regen-dictionary
```

2. Create a `.war` or `.ear` file using one of the `build-*` commands:

See the “Key PolicyCenter gwpc Commands” on page 61 in the *Installation Guide* for information on how to use these commands.

3. Copy this file to the application server. The target location of the file is dependent on the application server. If necessary (and it is almost always necessary if you change the data model), PolicyCenter runs the database upgrade tool during application start up.

Data Types

This topic describes the Guidewire data types, what they are, how to customize a data type, and how to create a new data type.

This topic includes:

- “Overview of Data Types” on page 233
- “The Data Types Configuration File” on page 236
- “Customizing Base Configuration Data Types” on page 237
- “Working with the Medium Text Data Type (Oracle)” on page 239
- “The Data Type API” on page 239
- “Defining a New Data Type: Required Steps” on page 241
- “Defining a New Tax Identification Number Data Type” on page 241

See also

- “Monetary Amounts in the Data Model and in Gosu” on page 111 in the *Globalization Guide*

Overview of Data Types

In the Guidewire data model, a *data type* is an augmentation of an object property, along three axes:

Axis	Description
Constraint	A data type can restrict the range of allowable values. For example, a String data type can restrict values to a maximum character limit.
Persistence	A data type can specify how PolicyCenter stores a value in the database and in the object layer. For example, one String data type can store values as CLOB (Character Large Object) objects. Another String data type can store values as VARCHAR objects.
Presentation	A data type can specify how the PolicyCenter interface treats a value. For example, a String data type can specify an input mask to use in assisting the user with data entry.

Guidewire stores the definitions for the base configuration data types in *.dti files in the datatypes directory. Each file corresponds to a separate data type, which the file name specifies.

Every data type has an associated Java or Gosu type (defined in the valueType attribute). For example, the associated type for the datetime data type is java.util.Date. Thus, you see the following XML code in the datetime.dti file.

```
<DataTypeDef xmlns="http://guidewire.com/datatype"
    type="com.guidewire.pl.metadata.datatype2.impl.DateTimeTypeDef"
    valueType="java.util.Date">
    ...

```

In a similar manner, the decimal data type has an associated type of java.math.BigDecimal.

```
<DataTypeDef xmlns="http://guidewire.com/datatype"
    type="com.guidewire.pl.metadata.datatype2.impl.DecimalTypeDef"
    valueType="java.math.BigDecimal">
    ...

```

Working with Data Types

In working with data types, you can do the following:

Operation	Description
Customize an existing data type	Modify the data type definition in file datatypes.xml, which you access through Studio. You can modify only a select subset of the base configuration data types. See “Customizing Base Configuration Data Types” on page 237.
Create a new data type	Create a .dti definition file and place it in PolicyCenter/modules/configuration/config/datatypes. You also need to create Gosu code to manage the data type. See “Defining a New Data Type: Required Steps” on page 241.
Override the data type on a column	Override the parameterization of the data type on individual columns (fields) on an entity. For example, you can make a VARCHAR column in the base data model use encryption by extending the entity and setting the encryption parameter on a <columnParam> element.

Using Data Types

You can use any of the data types for data fields (except for those that Guidewire reserves for itself). This includes data types that are part of the base configuration or data types that you create yourself. If you add a new column (field) to an entity or create a new entity, then you can use any data type that you want for that entity field. You do this by setting the type attribute on the column. For example:

```
<extension entityName="Policy">
    <column name="NewCompanyName" type="CompanyName" nullok="true" desc="Name for the new company."/>
</extension>
```

If you add too many large fields to any one table, you can easily reach the maximum row size of a table. In particular, this is a problem if you add a large number of long text or VARCHAR fields. Have your company database administrator (DBA) determine the maximum row size and increase the page size, if needed.

Guidewire-Reserved Data Types

Guidewire reserves the right to use the following data types exclusively. Guidewire does not support the use of these data types except for its own internal purposes. Do not attempt to create or extend an entity using one of the following data types:

- foreignkey
- key
- typekey
- typelistkey

Database Data Types

Guidewire bases its base configuration data types on the following database data types:

- BIT
- BLOB
- CLOB
- DECIMAL
- INTEGER
- TIMESTAMP
- VARCHAR

Data Types and Database Vendors

It is possible to see both VARCHAR and varchar in the Guidewire documentation. This usage has the following meanings.

All Upper-case Characters

This refers to database data types generally, for example VARCHAR and CLOB (Character Large Object). Of the supported database vendors, the Oracle (and H2) databases use upper-case data type names, while the SQL Server database uses lower-case data type names. To view the entire set of database data types, consult the database vendor's documentation.

All Lower-case Characters

This refers to Guidewire data types generally, for example, varchar and text. You can determine the set of Guidewire data types by viewing the names of the data type metadata definition files (*.dti) in the following application locations:

config/datatypes

Defining a Data Type for a Property

Guidewire associates data types with object properties using the following annotation:

`gw.datatype.annotation.DataType`

The annotation requires you to provide the name of the data type, along with any parameters that you want to supply to the data type.

- You associate a data type with a metadata property by specifying the `type` attribute on the `<column>` element.
- You specify any parameters for the data type with `<columnParam>` elements, children of the `<column>` element.

At runtime, PolicyCenter translates these metadata elements into instances of the `gw.datatype.annotation.DataType` annotation on the property corresponding to the `<column>`.

Each data type has a value type. You can associate a data type only with a property that has a feature type that matches the data type of the value type. For example, you can only associate a `String` data type with `String` properties.

Note: Guidewire PolicyCenter does not enforce this restriction at compile time. (However, PolicyCenter does check for any exception to this restriction at application server start up.) Guidewire permits annotations on any allowed feature, as long as you supply the parameters that the annotation requires. Therefore, you need to be aware of this restriction and enforce it yourself.

The Data Types Configuration File

IMPORTANT You must perform a database upgrade if you make changes to the `datatypes.xml` file. You must increment the version number in `extensions.properties` (in PolicyCenter Studio) to force a database upgrade upon application server start-up.

PolicyCenter lets you modify certain attributes on a subset of the base configuration data types by using the `datatypes.xml` configuration file. You can access this file in Studio from `configuration` → `config` → `fieldvalidators`. You can modify the values of certain attributes in this file to customize how these data types work in PolicyCenter.

This `datatypes.xml` file contains the following elements:

XML element	Description
<code><DataTypes></code>	Top XML element for the <code>datatypes.xml</code> file.
<code><...DataType></code>	Subelement that defines a specific <i>customizable data type</i> (for example, <code>PhoneDataType</code> , <code>YearDataType</code> , <code>MoneyDataType</code>) and assigns one or more default values to each one.

WARNING Modify the `datatypes.xml` file with caution. If you modify the file incorrectly, you can invalidate your PolicyCenter installation.

`<...DataType>`

The `<...DataType>` element is the basic element of the `datatypes.xml` file. It assigns default values to base configuration data types that Guidewire permits you to customize. This element starts with the specific data type name. For example, the element for the `PercentageDec` data type in the `datatypes.xml` file is `<PercentageDecDataType>`.

The `<...DataType>` element has the following attributes:

Attribute	Description
<code>length</code>	Assigns the maximum character length of the data type.
<code>validator</code>	Binds the data type to a given validator definition. It must match the <code>name</code> attribute of the validator definition.
<code>precision</code>	Used for <code>DECIMAL</code> types only.
<code>scale</code>	<ul style="list-style-type: none">• <code>precision</code> is the total number of digits in the number.• <code>scale</code> is the number of digits to the right of the decimal point. The default value is 2. <p>The value of <code>scale</code> must be less than the value of <code>precision</code>.</p> <p>For more information, see “The Precision and Scale Attributes” on page 237.</p>
<code>appscale</code>	Optional attribute for use with <code>money</code> data types.
	For more information, see “The Money Data Type” on page 239.

Deploying Modifications to Data Types Configuration File

If you change the `datatypes.xml` file, then you need to deploy those changes to the application server. Most modifications to the `datatypes.xml` file take effect the next time the server reboots.

- PolicyCenter reloads the `validator` attribute for data type definitions upon server reboot. This is so that you can rebind different validators to data types.

- PolicyCenter does not reload other data type attributes such as `length`, `precision`, and `scale`. This is because PolicyCenter applies these attributes only during the initial server boot. (It uses them during table creation in the database.) PolicyCenter ignores any changes to these attributes unless something triggers a database upgrade. For example, if you modify a base entity, then PolicyCenter triggers a database upgrade at the next server restart.

Guidewire Recommendations for Modifying Data Types

Guidewire recommends the following:

- Make modifications to the data types before creating the PolicyCenter database for the first time.
- Make modifications to the data types before performing a database upgrade that creates a new extension column.

PolicyCenter looks at the data type definitions only at the time it creates a database column. Thus, it ignores any changes after that point. However, any differences between the type definition and the actual database column can cause upgrade errors or failure warnings. Therefore, Guidewire recommends that you exercise extreme caution in making changes to type definitions.

Customizing Base Configuration Data Types

You can customize the behavior of the data types listed in `datatypes.xml`. To see exactly what you can customize for each data type, see “List of Customizable Data Types” on page 238. In general, though, you can customize some or all of the following attributes on a listed data type (depending on the data type):

- `length`
- `precision`
- `scale`
- `validator`

The Length Attribute

Data types based on the `VARCHAR` data type have a `length` attribute that you can customize. This attribute sets the maximum allowable character length for the field (column).

The Precision and Scale Attributes

Data types based on the `DECIMAL` data type have `precision` and `scale` attributes that you can customize. These attributes determine the size of the decimal. The `precision` value sets the total number of digits in the number and the `scale` value is the number of digits to the right of the decimal point.

There are special requirements for these attributes in working with monetary amounts. For more information, see “Precision and Scale of Monetary Amounts” on page 112 in the *Globalization Guide*.

The Validator Attribute

Most data types have a `validator` attribute that you can customize. This attribute binds the data type to a given validator definition. For example, `PhoneDataType` (defined in `datatypes.xml`) binds to the `Phone` validator by its `validator` attribute. This matches the `name` attribute of a `<ValidatorDef>` definition in file `fieldvalidators.xml`.

```
//File datatypes.xml
<DataTypes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="../../../../../../platform/p1/xsd/datatypes.xsd">
    ...
    <PhoneDataType length="30" validator="Phone"/>
    ...
</DataTypes>

//File fieldvalidators.xml
<FieldValidators>
```

```

...
<ValidatorDef description="Validator.Phone" input-mask="###-###-### x###" name="Phone"
    value="[0-9]{3}-[0-9]{3}-[0-9]{4}( x[0-9]{0,4})?" />
...
</FieldValidators>

```

See also

- For information on field validators in general, see “Field Validation” on page 259.
- For information on how to localize field validation, see “Configuring National Field Validation” on page 165 in the *Globalization Guide*.

List of Customizable Data Types

The following table summarizes the list of the data types that you can customize. PolicyCenter defines these data types in `datatypes.xml`. If a data type does not exist in `datatypes.xml`, then you cannot customize its attributes.

PolicyCenter builds the all of its data types on top of the base database data types of CLOB, TIMESTAMP, DECIMAL, INTEGER, VARCHAR, BIT, and BLOB.

Note: CLOB stands for Character Large Object and represents character data of a very large size, typically 2GB or more. BLOB stands for Binary Large Object or Basic Large Object and represents binary data.

Note: Only decimal numbers use the `precision` and `scale` attributes. The `precision` attribute defines the total number of digits in the number. The `scale` attribute defines the number of digits to the right of the decimal point. Therefore, `precision` must be greater than or equal to `scale`.

Guidewire data type	Built on	Customizable attributes
ABContactMatchSetKey	VARCHAR	length
Account	VARCHAR	length, validator
AddressLine	VARCHAR	length, validator
ClaimNumber	VARCHAR	length, validator
CompanyName	VARCHAR	length, validator
ContactIdentifier	VARCHAR	length, validator
CreditCardNumber	VARCHAR	length, validator
DaysWorkedWeek	DECIMAL	precision, validator
DriverLicense	VARCHAR	length, validator
DunAndBradstreetNumber	VARCHAR	length, validator
EmploymentClassification	VARCHAR	length, validator
ExchangeRate	DECIMAL	precision, validator
Exmod	DECIMAL	precision, validator
FirstName	VARCHAR	length, validator
HoursWorkedDay	DECIMAL	precision, validator
LastName	VARCHAR	length, validator
MediumText	VARCHAR	length
Money	DECIMAL	precision, app, validator
PercentageDec	DECIMAL	precision
Phone	VARCHAR	length, validator
PolicyNumber	VARCHAR	length, validator
PostalCode	VARCHAR	length, validator
ProrationFactor	DECIMAL	precision, validator
Rate	DECIMAL	precision, validator
RatingLineBasisAmount	DECIMAL	precision, validator
Risk	DECIMAL	precision, validator
Speed	INTEGER	validator

Guidewire data type	Built on	Customizable attributes
SSN	VARCHAR	length, validator
VIN	VARCHAR	length, validator
Year	INTEGER	validator

The Percentage Decimal Data Type

Guidewire builds the PercentageDec data type on top of the DECIMAL (3,0) data type. Only use decimal values from 0 to 100 inclusive.

The Money Data Type

Guidewire provides the Money data type as the basis for the <monetaryamount> subelement in metadata definition files. The <monetaryamount> subelement is a compound field type, with a Money data type as one component and a typekey to the Currency typelist as the other component.

For more information, see “Monetary Amounts in the Data Model and in Gosu” on page 111 in the *Globalization Guide*.

Working with the Medium Text Data Type (Oracle)

In working with the MEDIUMTEXT data type, take extra care if you use multi-byte characters, excluding CLOB-based data types such as LONGTEXT, TEXT, or CLOB in the Oracle database. (CLOB stands for Character Large OBject.) On Oracle, Guidewire supports any single-byte character set, or the multi-byte character sets UTF8 and AL32UTF8.

Oracle has a maximum column width, for non-LOB columns, of 4000 bytes. Thus, with a single-byte character set, you can store up to 4000 characters in a single column (because one character requires one byte). However, with a multi-byte character set, you can store fewer characters, depending on the ratio of bytes to characters for that character set. For UTF8, the ratio is at most three-to-one, so you can always safely store up to $4000 / 3 = 1333$ characters in a single column.

Thus, Guidewire recommends:

- Limit the number of characters to 4000 if using a single-byte character set.
- Limit the number of characters to 1333 if using UTF8 or AL32UTF8. However, it is possible that some AL32UTF8 characters can be four bytes, and thus 1333 of them can potentially overflow 4000 bytes.

The Data Type API

The classes in `gw.datatype` form the core of the Data Type API. Most of the time, you do not need to use data types directly, as Guidewire uses these internally in the system. However, there can be cases in which you need to access a data type, typically to determine the constraints information.

This topic includes:

- Retrieving the Data Type for a Property
- Retrieving a Particular Data Type in Gosu
- Retrieving a Data Type Reflectively
- Using the `IDataType` Methods

Retrieving the Data Type for a Property

To retrieve the data type for a property, you could look up the annotation on the property. You could then look up the data type reflectively, using the `name` and `parameters` properties of the annotation. However, this is a cumbersome process. As a convenience, use the following method instead:

```
gw.datatype.DataTypes.get(gw.lang.reflect.IAnnotatedFeatureInfo)
```

For example:

```
var property = Claim.Type.TypeInfo.getProperty("ClaimNumber")
var claimNumberDataType = DataTypes.get(property)
```

The `gw.datatype.DataTypes.get(gw.lang.reflect.IAnnotatedFeatureInfo)` method also provides some performance optimizations. Therefore, Guidewire recommends that you use this method rather than looking up the annotation directly from the property.

Retrieving a Particular Data Type in Gosu

If you need an instance of a particular data type, use the corresponding method on `gw.datatype.DataTypes`. A static method exists on this type for each data type in the system. Some data types have two methods:

- One method that takes all parameters
- One method that takes only the required parameters

For example:

```
var varcharDataType = DataTypes.varchar(10)
var encryptedVarcharDataType = DataTypes.varchar(10,
    /* validator */ null,
    /* logicalSize */ null,
    /* encryption */ true,
    /* trimwhitespace */ null)
```

Retrieving a Data Type Reflectively

In rare cases, you may need to look up a data type reflectively. To do this, you need the name of the data type, and a map containing the parameters for the data type. For example:

```
var varcharDataType = DataTypes.get("varchar", { "size" -> "10" })
```

Using the `IDataType` Methods

After you have a data type, you can access its various aspects using one of the `asXXXDataType` methods, which are:

- `asConstrainedDataType()` : `IConstrainedDataType`
- `asPersistentDataType()` : `IPersistentDataType`
- `asPresentableDataType()` : `IPresentableDataType`

For example, suppose that you want to determine the maximum length of a property:

```
var claim : Claim = ...
var claimNumberProperty = Claim.Type.TypeInfo.getProperty("ClaimNumber")
var claimNumberDataType = DataTypes.get(claimNumberProperty)
var maxLength = claimNumberDataType.asConstrainedDataType().getLength(claim, claimNumberProperty)
```

It may seem odd that the `getLength(java.lang.Object, gw.lang.reflect.IPropertyInfo)` method (in this example) takes the claim and the claim number property. The reason for this is that the constraint and presentation aspects of data types are dynamic, meaning that they are based on context.

Many of the methods on `gw.datatype.IConstrainedDataType` and `gw.datatype.IPresentableDataType` take a context object, representing the owner of the property with the data type, along with the property in question. This allows the implementation to provide different behavior, based on the context. If you do not have the context object or property, then you can pass `null` for either of these arguments.

If you implement a data type, then you must handle the case in which the context is unknown.

Defining a New Data Type: Required Steps

The process of defining a new data type requires multiple steps.

1. Register the data type within Guidewire PolicyCenter by creating a `.dti` file (data type declaration file). To do this in Studio:

- a. In the Project window, navigate to **configuration** → **config** → **datatypes**.
- b. Right-click **datatypes**, and then click **New** → **File**.
- c. Enter the name of the data type to name the file. You must add the `.dti` extension. Studio does not do this for you. Studio inserts this file in the correct location.
- d. The first time that you do this, you are prompted to create a new file type association for `*.dti` files. In the **Register New File Type Association** dialog, click **Open matching files in Studio**, and then in the list under that option click **Text files**. Click **OK**.

You must enter definitions for the following items for the data type. If necessary, view other samples of data-type definition files to determine what you need to enter.

- Name
- Value type
- Parameters
- Implementation type

2. Create a data type definition class that implements the `gw.datatype.def.IDataTypeDef` interface. This class must include writable property definitions that correspond to each parameter that the data type accepts.
3. Create data type handler classes for each of the three aspects of the data type (constraints, persistence, and presentation). These classes must implement the following interfaces:
 - `gw.datatype.handler.IDataTypeConstraintsHandler`
 - `gw.datatype.handler.IDataTypePersistenceHandler`
 - `gw.datatype.handler.IDataTypePresentationHandler`

Guidewire provides a number of implementations of these three interfaces for the standard data types. For example, you can create your own CLOB-based data types by defining a data type that uses the `ClobPersistenceHandler` class. To access the handler interface implementations or to view a complete list, enter the following within Gosu code:

```
gw.datatypes.impl.*
```

After you create the data type, you will want to use the data type in some useful way. For example, you can create an entity property that uses that data type and then expose that property as a field within PolicyCenter.

See also

- For a discussion of constraints, persistence, and presentation as it relates to data types, see “Overview of Data Types” on page 233.

Defining a New Tax Identification Number Data Type

The following examples illustrates the steps involved in defining a new data type and using it. The example defines a new data type for *Tax Identification Number* objects, called `TaxID`. The data type has one required property, the name of the property on the context object. This property, `countryProperty`, identifies which country is in context for validating the data.

This example contains the following steps:

- Step 1: Register the Data Type

- Step 2: Implement the `IDataTypeDef` Interface
- Step 3: Implement the Data Type Aspect Handlers

Step 1: Register the Data Type

To register a new data type, create a file named `xxx.dti`, with `xxx` as the name of the new data type. In this case, create a file named `TaxID.dti`. To do this:

1. In the Project window, navigate to `configuration → config → datatypes`.
2. Right-click `datatypes`, and then click `New → File`.
3. Enter `TaxID.dti` as the file name. This action creates an empty data type file and places it in the `datatypes` folder.
4. Enter the following text in the file:

```
<?xml version="1.0"?>
<DataTypeDef xmlns="http://guidewire.com/datatype" type="gw.newdatatypes.TaxIDDataTypeDef"
    valueType="java.lang.String">
    <ParameterDef name="countryProperty" desc="The name of a property on the owning entity,
        whose value contains the country with which to validate and format values."
        required="true" type="java.lang.String"/>
</DataTypeDef>
```

The root element of `TaxID.dti` is `<DataTypeDef>` and the namespace is `http://guidewire.com/datatype`.

This example defines the following:

data type name	<code>TaxID</code>
value type	<code>String</code>
parameter	<code>contactType</code>
implementation type	<code>gw.newdatatypes.TaxIDDataTypeDef</code>

See also

- For details on the attributes and elements relevant to the data type definition, see “The PolicyCenter Data Model” on page 149.

Step 2: Implement the `IDataTypeDef` Interface

The implementation class that you create to handle the `TaxID` data type must do the following:

- It must implement the `gw.datatype.def.IDataTypeDef` interface.
- It must have a no-argument constructor.
- It must have a property for each of the data type parameters.

For example, suppose that you have a new data type that has a `String` parameter named `someParameter`. The implementation class (specified in the `type` attribute) must define a writable property named `someParameter`, so that the data type factory can pass the argument values to the implementation. The implementation can then use the parameters in the implementation of the various handlers, which are:

- `gw.datatype.handler.IDataTypeConstraintsHandler`
- `gw.datatype.handler.IDataTypePersistenceHandler`
- `gw.datatype.handler.IDataTypePresentationHandler`

Class `TaxIDDataTypeDef`

For our example data type, the `gw.newdatatypes.TaxIDDataTypeDef` class looks similar to the following. To create this file, first create the package, then the class file, in the Studio `Classes` folder.

```
package gw.newdatatypes
uses gw.datatype.def.IDataTypeDef
```

```
uses gw.datatype.handler.IDataTypeConstraintsHandler
uses gw.datatype.handler.IDataTypePresentationHandler
uses gw.datatype.handler.IDataTypePersistenceHandler
uses gw.lang.reflect.IPropertyInfo
uses gw.datatype.handler.IDataTypeValueHandler
uses gw.datatype.def.IDataTypeDefValidationErrors
uses gw.datatype.impl.VarcharPersistenceHandler
uses gw.datatype.impl.SimpleValueHandler

class TaxIDDataTypeDef implements IDatatypeDef {
    private var _countryProperty : String as CountryProperty

    override property get ConstraintsHandler() : IDatatypeConstraintsHandler {
        return new TaxIDConstraintsHandler(CountryProperty)
    }

    override property get PersistenceHandler() : IDatatypePersistenceHandler {
        return new VarcharPersistenceHandler(/* encrypted */ false,
                                             /* trimWhitespace */ true,
                                             /* size */ 30)
    }

    override property get PresentationHandler() : IDatatypePresentationHandler {
        return new TaxIDPresentationHandler(CountryProperty)
    }

    override property get ValueHandler() : IDatatypeValueHandler {
        return new SimpleValueHandler(String)
    }

    override function validate(prop : IPropertyInfo, errors : IDatatypeDefValidationErrors) {
        // Check that the CountryProperty names an actual property on the owning type, and that
        // the type of the property is typekey.Country.
        var countryProp = prop.OwnersType.TypeInfo.getProperty(CountryProperty)

        if (countryProp == null) {
            errors.addError("Property '" + CountryProperty + "' does not exist on type " +
                           prop.OwnersType)
        } else if (not typekey.Country.Type.isAssignableFrom(countryProp.Type)) {
            errors.addError("Property " + countryProp + " does not resolve to a " + typekey.Country)
        }
    }
}
```

Note that the class defines a property named `CountryProperty`, which the system calls to pass the `countryProperty` parameter. Also notice how the implementation reads the value of `CountryProperty` as its constructs its constraints and presentation handlers. Guidewire guarantees to fill the implementation parameters before calling the handlers.

In the example code, the class refers to constraints and presentation handlers created specifically for this data type. However, it also reuses a Guidewire-provided persistence handler, the `VarcharPersistenceHandler`. You do not usually need to create your own persistence handler, as Guidewire defines persistence handlers for all the basic database column types.

Step 3: Implement the Data Type Aspect Handlers

As you define a new data type, it is possible (actually likely) that you need to define one or more handlers for the data type. These handler interfaces are different than the Data Type API interfaces. For example, clients that use the Data Type API use the following:

```
gw.datatype.IConstrainedDataType
```

However, if you define a new data type, you must implement the following:

```
gw.datatype.handler.IDataTypeConstraintsHandler
```

This separation of interfaces allows the definition of a caller-friendly interface for data type clients and a implementation-friendly interface for data type designers.

The example data type defines a handler for both constraints and presentation.

Class TaxIDConstraintsHandler

This class looks similar to the following:

```
package gw.newdatatypes

uses gw.datatype.handler.IStringConstraintsHandler
uses gw.lang.reflect.IPropertyInfo
uses java.lang.Iterable
uses java.lang.Integer
uses java.lang.CharSequence
uses gw.datatype.DataTypeException

class TaxIDConstraintsHandler implements IStringConstraintsHandler {

    var _countryProperty : String

    construct(countryProperty : String) {
        _countryProperty = countryProperty
    }

    override function validateValue(ctx : Object, prop : IPropertyInfo, value : Object) {
        var country = getCountry(ctx)

        switch (country) {
            case "US": validateUSTaxID(ctx, prop, value as java.lang.String)
                break
            // other countries ...
        }
    }

    override function validateUserInput(ctx : Object, prop : IPropertyInfo, strValue : String) {
        validateValue(ctx, prop, strValue)
    }

    override function getConsistencyCheckerPredicates(columnName : String) : Iterable<CharSequence> {
        return {}
    }

    override function getLoaderValidationPredicates(columnName : String) : Iterable<CharSequence> {
        return {}
    }

    override function getLength(ctx : Object, prop : IPropertyInfo) : Integer {
        var country = getCountry(ctx)

        switch (country) {
            case "US": return ctx typeis Person ? 11 : 10
            // other countries ...
        }

        return null
    }

    private function getCountry(ctx : Object) : Country {
        return ctx[_countryProperty] as Country
    }

    private function validateUSTaxID(ctx : Object, prop : IPropertyInfo, value : String) {
        var pattern = ctx typeis Person ? "\d{3}-\d{2}-\d{4}" : "\d{2}-\d{7}"
        if (not value.matches(pattern)) {
            throw new DataTypeException("${value} does not match required pattern ${pattern}", prop,
                "Validation.TaxID", { value })
        }
    }
}
```

Class TaxIDPresentationHandler

This class looks similar to the following:

```
package gw.newdatatypes

uses gw.lang.reflect.IPropertyInfo
uses gw.datatype.handler.IStringPresentationHandler
```

```
class TaxIDPresentationHandler implements IStringPresentationHandler {  
    private var _countryProperty : String  
  
    construct(countryProperty : String) {  
        _countryProperty = countryProperty  
    }  
  
    function getEditorValue(ctx : Object, prop : IPropertyInfo) : Object {  
        return null  
    }  
  
    override function getDisplayFormat(ctx : Object, prop : IPropertyInfo ) : String {  
        return null  
    }  
  
    override function getInputMask(ctx : Object, prop : IPropertyInfo) : String {  
  
        switch (getCountry(ctx)) {  
            case "US": return ctx.typeis Person ? "###-##-####" : "##-#####"  
            // other countries ...  
        }  
  
        return null  
    }  
  
    override function getPlaceholderChar(ctx : Object, prop : IPropertyInfo) : String {  
        return null  
    }  
  
    private function getCountry(ctx : Object) : Country {  
        return ctx[_countryProperty] as Country  
    }  
}
```

Notice how each of these handlers makes use of the context object in order to determine the type of input mask and validation string to use.

The Archiving Domain Graph

The archiving domain graph in PolicyCenter defines which entity types are included in archived policy periods.

IMPORTANT Guidewire strongly recommends that you contact Customer Support before implementing archiving.

This topic includes:

- “Domain Graph Overview” on page 247
- “Object Ownership in the Domain Graph” on page 248
- “Accessing the Domain Graph” on page 250
- “Including Objects in the Domain Graph” on page 251
- “Domain Graph Validation” on page 255
- “Working with Changes to the Data Model” on page 257
- “Working with Shared Entity Data” on page 257
- “Eliminating Cycles From the Domain Graph” on page 258

See also

- For a complete list of topics related to archiving, see “More Information on Archiving” on page 325 in the *Application Guide*.

Domain Graph Overview

The *domain graph* defines the cluster of related entity instances that PolicyCenter treats as a single aggregate for purposes of archiving data. The domain graph begins with a *root* entity type and ends at a *boundary* of related entity types.

- The *root* of the domain graph is a specific entity type. Starting with the root entity type, the graph follows relationships in the data model to other entity types until the boundary of the graph is reached.

In PolicyCenter, the root entity type in the domain graph is `PolicyPeriod`.

- The *boundary* defines which entity types terminate the traversal of relationships from the root and thus prescribe the extent of entities within the graph.

In PolicyCenter, the boundary includes the major entities that relate to the `PolicyPeriod` entity type, including effective dated entity types such as `PolicyAddress` and `PolicyDriver`. The boundary also includes other entity types, such as `Job`, and `Note`.

Entity types with foreign key relationships to the root entity type and that implement the `Extractable` delegate are included in the domain graph.

See also

- “Including Objects in the Domain Graph” on page 251

The Domain Graph Is a Directed Acyclic Graph

The domain graph is an example of a *directed acyclic graph*. It is a *graph* that comprises nodes and relationships between them. It is a *directed* graph, because you can traverse a relationship in one direction only. For example, if node A and node B have a relationship, you can traverse from A to B or from B to A but not in both directions. It is a directed *acyclic* graph, because traversal of cycles in the graph is not permitted. For example, if you can traverse from node A to node B and from node B to node C, you then cannot traverse from C to A.

The PolicyCenter data model itself also is a directed acyclic graph based on foreign key relationships between entity types. Foreign key cycles in the data model graph are prohibited so PolicyCenter can commit related entity instances in a transactional bundle in a safe order without exceptions or deadlocks.

See also

- “Domain Graph Validation” on page 255
- “Eliminating Cycles From the Domain Graph” on page 258

The Domain Graph and Object Graphs in the Database

The domain graph is a subset of the data model graph and begins with the root entity, `PolicyPeriod`. Instances of the domain graph are instances of the root entity and *object graphs* of their related entities that implement the `Extractable` delegate. An object graph comprises an entity instance and the specific entity instances related to it. The PolicyCenter data model has just one domain graph. The PolicyCenter database has as many instances of the domain graph as there are instances of the root entity.

Each instance of the domain graph – the object graph of a specific root entity instance – can be archived. Some instances of the domain graph are complete, with all entity instances from the root to the boundaries included. Others are less complete, because the object graphs of those root entities do not include all possibly related entity types. For example, the domain graph may include notes and activities, but a specific instance of the root entity has no notes attached or activities assigned.

Object Ownership in the Domain Graph

The relationships that the domain graph captures are that of ownership.

Ownership in the Domain Graph Through Foreign Keys

Guidewire defines *ownership* of one object by another by a foreign key between them. In general, a foreign key to object A from object B indicates that object A *owns* object B. Generally, foreign keys point from owned objects to owning objects in the domain graph.

In the following example XML metadata definition, the foreign key field RootID points to the owner of the entity type that contains the foreign key.

```
<foreignkey name="RootID"
    fkentity="TestGraphRoot"
    desc="The owner of this entity type in the domain graph"/>
```

In the preceding example, the owner is the entity type TestGraphRoot.

For example, Note and PolicyPeriod have an ownership relationship. In the base configuration, Note has a foreign key to PolicyPeriod. This foreign key by itself indicates that PolicyPeriod instances own Note instances.

Note: A Note also can be owned by other entities, such as Job.

The following Note metadata definition extension illustrates a foreign key that defines an ownership relationship.

```
<internalExtension xmlns="http://guidewire.com/datamodel"
    entityName="Note" javaClass="com.guidewire.pc.domain.note.Note">
    ...
    <foreignkey columnName="PolicyPeriodID"
        desc="Associated Policy Period."
        exportable="false"
        fkentity="PolicyPeriod"
        name="PolicyPeriod"/>
    ...
</internalExtension>
```

In the preceding example, the owner of the Note entity is the PolicyPeriod entity.

Inverse Ownership in the Domain Graph

In some cases, an owning object has a foreign key that points to an owned object. This direction of a foreign key is the inverse of typical ownership relationships in the domain graph. To indicate this inverse direction of ownership, you set the owner attribute on the foreign key to true.

```
<foreignkey name="OwnedObjectID"
    fkentity="TestGraphOwnedObject"
    owner="true"/>
```

In the preceding example, the entity type or entity type extension with the foreign key definition is the owner of the TestGraphOwnedObject entity type.

Note: Guidewire discourages the use of inverse ownership relationships. The PolicyCenter data model supports inverse ownership relationships for the rare case in which upgrading the database is unduly cumbersome or time consuming. Do not use this type of relationship as a general rule.

See also

- “Eliminating Cycles From the Domain Graph” on page 258

Ownership Through the Effective Dated Branch

In PolicyCenter, an effective dated entity is owned by the root of the effective dated branch. For example, PolicyLocation is an effective dated entity owned by PolicyPeriod.

The following PolicyLocation metadata definition illustrates how an effective dated entity is owned by PolicyPeriod.

```
<entity
    ...
    effDatedBranchType="PolicyPeriod"
    entity="PolicyLocation"
    ...
    type="effdated">
```

Accessing the Domain Graph

Guidewire uses the DOT plain text graph description language to represent the domain graph. The DOT language describes complex graph relationships in a way that both humans and computers can use. Files that contain DOT text generally end with a .dot extension. Specialized software can generate a visual presentation of the domain graph from DOT text.

Guidewire provides access to the DOT text description of the domain graph through the **Server Tools** menu, which anyone with system administration privileges can access. In addition, archiving must be enabled to access the domain graph.

The **Server Tools** → **Info Pages** → **Domain Graph Info** screen contains the following cards:

- **Graph** – Provides a human-readable text version of the domain graph. Also provides a **Download** button, which generates a ZIP file of .dot files that contain the text version of the domain graph.
- **Warnings** – Provides a list of issues with the domain graph found when the server started and that can lead to errors in archiving. Guidewire strongly recommends that you review the **Warnings** card any time you change the data model.

See also

- For information on enabling archiving, see “Archiving-related Configuration Parameters” on page 452.
- “Using the Server Tools” on page 133 in the *System Administration Guide*.

Viewing the Textual Domain Graph

In the default configuration, you can view the domain graph in DOT text notation.

To access the textual domain graph

1. Log into PolicyCenter by using an administrative account.
2. Press ALT+SHIFT+T to display the **Server Tools** tab.
3. In the left sidebar, select **Info Pages** → **Domain Graph Info**.

From this screen, you can:

- View the text version of the domain graph.
- Download a ZIP file that contains .dot files that contain the text version of the domain graph.
- View issues on the **Warnings** card that PolicyCenter generated during validation of the domain graph when the server started.

Viewing the Visual Domain Graph

To view a visual presentation of the domain graph, you must download and install specialized software that can read .dot files and render an image from DOT text notation. There are many software programs available that you can use for this purpose, some of which are open source or free.

One such program is Graphviz, which you can download from the following URL:

<http://www.graphviz.org/About.php>

Add the path for a software executable that renders graph images from DOT text notation to the PATH environment variable. Then, the ZIP file that you download from the **Domain Graph Info** screen includes a PDF file with a visual presentation of the domain graph. You can generate graph images in other file formats from the command line.

To view the visual domain graph

1. Download and install software that can read .dot files and render graph images from those files.

2. Navigate to Server Tools.
 3. In the menu on the left, click Info Pages.
 4. Select Domain Graph Info from the drop-down list at the top of the screen.
 5. Click Download, and save the ZIP file to your local machine.
 6. Extract the contents of the ZIP file into a permanent directory.

If you added the software executable to the PATH environment variable, the download file includes a visual presentation of the domain graph in a PDF file.

Complete the following steps only if you want a visual presentation of the domain graph in a file format other than PDF.
 7. Open a command window and navigate to the directory into which you extracted the .dot files.
 8. Enter the following at the command prompt:

```
dot.exe -Tpng -o<graphic_file_name.png> <DOT_file_name>
```

This Graphviz command creates a graphics file in PNG format that you can open in a graphic viewer. The graphic is quite large.
- Note:** This step assumes that you downloaded Graphviz. If you downloaded a different program, the command syntax that it supports.

Including Objects in the Domain Graph

For PolicyCenter to consider an object for archiving, the object must meet all of the following criteria:

- The object must implement a specific delegate, depending on the object purpose.
- The object must have an ownership relationship with another object both within the graph.

The following table summarizes the object types and the delegates that each object type must implement.

Object	Implements...
RootInfo object	For PolicyCenter, the RootInfo object is PolicyPeriod. Only one object can implement the RootInfo delegate. You cannot change the RootInfo object.
All other domain objects	All domain objects, including the root object, must implement the Extractable delegate. In PolicyCenter, effective dated objects implement the Extractable delegate by inheritance through the EffDatedBase entity.
Reference objects	A reference object is a data object that multiple instances of a domain graph object share. In PolicyCenter, multiple PolicyPeriod objects can share the same User data.
Overlap table objects	Overlap tables are tables in which individual table rows can exist either in the domain graph or as part of reference data, but not both. The database table itself exists in both the domain graph and as reference data. In PolicyCenter, the Note and Document tables are overlap tables. The primary use for these types of objects is for Guidewire code. Their use by non-Guidewire code is not common. All overlap table objects, and overlap table objects only, must implement the OverlapTable delegate.

In addition, any new object that you want to add to the archiving domain graph must have an ownership relationship with another object already in the graph. You can establish ownership by defining a foreign key in the new object to another object already in the graph. Or, you can establish ownership by defining a foreign key to the new object from an object already in the graph and setting the attribute `owner="true"`. For effective dated entities, ownership is added indirectly through the effective dated branch.

See also

- “Implementing the Correct Delegate” on page 252
- “Defining Ownership Relations Between Objects” on page 254
- “Ownership Through the Effective Dated Branch” on page 249

Implementing the Correct Delegate

The archiving process requires certain entity types to implement one or more of the following delegates to ensure that instances of the domain graph can be archived successfully.

- `RootInfo` – See “The RootInfo Delegate” on page 252
- `Extractable` – See “The Extractable Delegate” on page 253
- `OverlapTable` – See “The OverlapTable Delegate” on page 253

A delegate entity type is a reusable entity that contains an interface and a default implementation of that interface. A delegate may also add its own columns to the tables of entities that implement the delegate. This type of delegation enables an entity type to implement an interface while delegating the implementation to the delegate.

WARNING Do not change the archiving delegates that Guidewire base entities implement to alter the archiving domain graph. For example, do not change which entity type implements the `RootInfo` delegate or change which base entity types implement the `Extractable` and `OverlapTable` delegates.

See also

- “Delegate Data Objects” on page 161

The RootInfo Delegate

Whenever PolicyCenter archives an instance of the domain graph, it leaves behind in the PolicyCenter database an entity instance that provides the following:

- Sufficient information for a minimal search on archived policy periods
- Sufficient information to retrieve the date.

This entity type must implement the `RootInfo` delegate. No other entity type may implement the `RootInfo` delegate.

In PolicyCenter, the root information entity type is the `PolicyPeriod` entity. You cannot change which entity type implements the `RootInfo` delegate. It must always be `PolicyPeriod`. PolicyCenter does not archive the `PolicyPeriod` table. PolicyCenter archives only the tables that the `PolicyPeriod` entity type owns.

IMPORTANT Because PolicyCenter does not archive the `PolicyPeriod` table, the table continues to grow regardless of archiving. Therefore, Guidewire recommends against extending `PolicyPeriod` to store large amounts of data, such as BLOB fields.

The Extractable Delegate

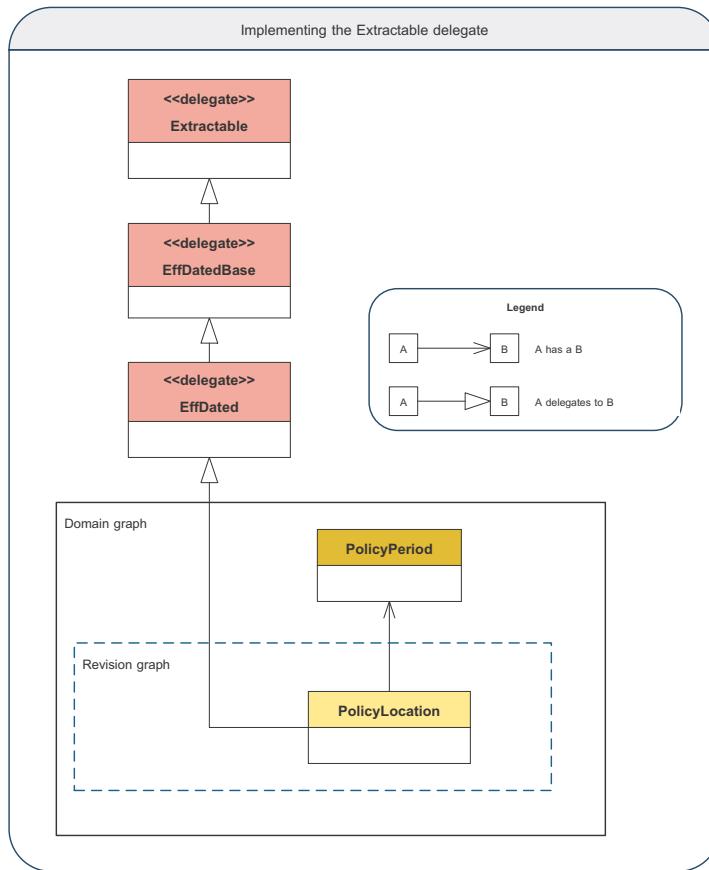
All entities in the archiving domain graph must implement the `Extractable` delegate. The converse is also true. No entity outside the domain graph may implement the `Extractable` delegate. The use of this delegate ensures the creation of the `ArchivePartition` column that PolicyCenter uses during the archive process.

The following metadata definition of the `Document` object shows that it implements the `Extractable` delegate:

```
<internalExtension
  xmlns="http://guidewire.com/datamodel"
  entityName="Document"
  ...
  ...
  <implementsEntity name="Extractable"/>
  ...
```

In addition, if you add an edge foreign key to an entity that is part of the domain graph, the edge foreign key must also implement the `Extractable` delegate. For example, if you create a subtype of `Contact`, then it must implement the `Extractable` delegate. The edge foreign key does not inherit the `<implementsEntity>` delegate from the enclosing entity. If you do not define the edge foreign key to implement the `Extractable` delegate, the application server refuses to start.

In PolicyCenter, entities in the revision graph implement the `Extractable` delegate through the `EffDated` delegate. The following illustration shows how `PolicyLocation` implements `Extractable`.



The OverlapTable Delegate

Overlap tables are tables with rows that can exist either in the domain graph or as part of reference data, but not both. However, individual rows in the table exist in either the domain graph or the reference data. Any attempt to create a table row that exists in both the domain graph and the reference data causes archiving to fail.

In PolicyCenter, the Document and Note tables are overlap tables. Therefore, the Document and Note tables can exist in either the domain graph or as part of reference data.

Objects that use overlap tables must implement the OverlapTable delegate. Implementing the OverlapTable delegate creates an additional Admin column that PolicyCenter uses to determine which rows belong to the domain graph and which do not.

Because these objects are both inside and outside the domain graph, they must also implement the Extractable delegate.

The following metadata definition of the Note object illustrates the use of multiple delegate implementations:

```
<internalExtension
  xmlns="http://guidewire.com/datamodel"
  entityName="Note"
  ...>
<implementsEntity
  name="Extractable"/>
<implementsEntity
  name="OverlapTable"/>
```

Defining Ownership Relations Between Objects

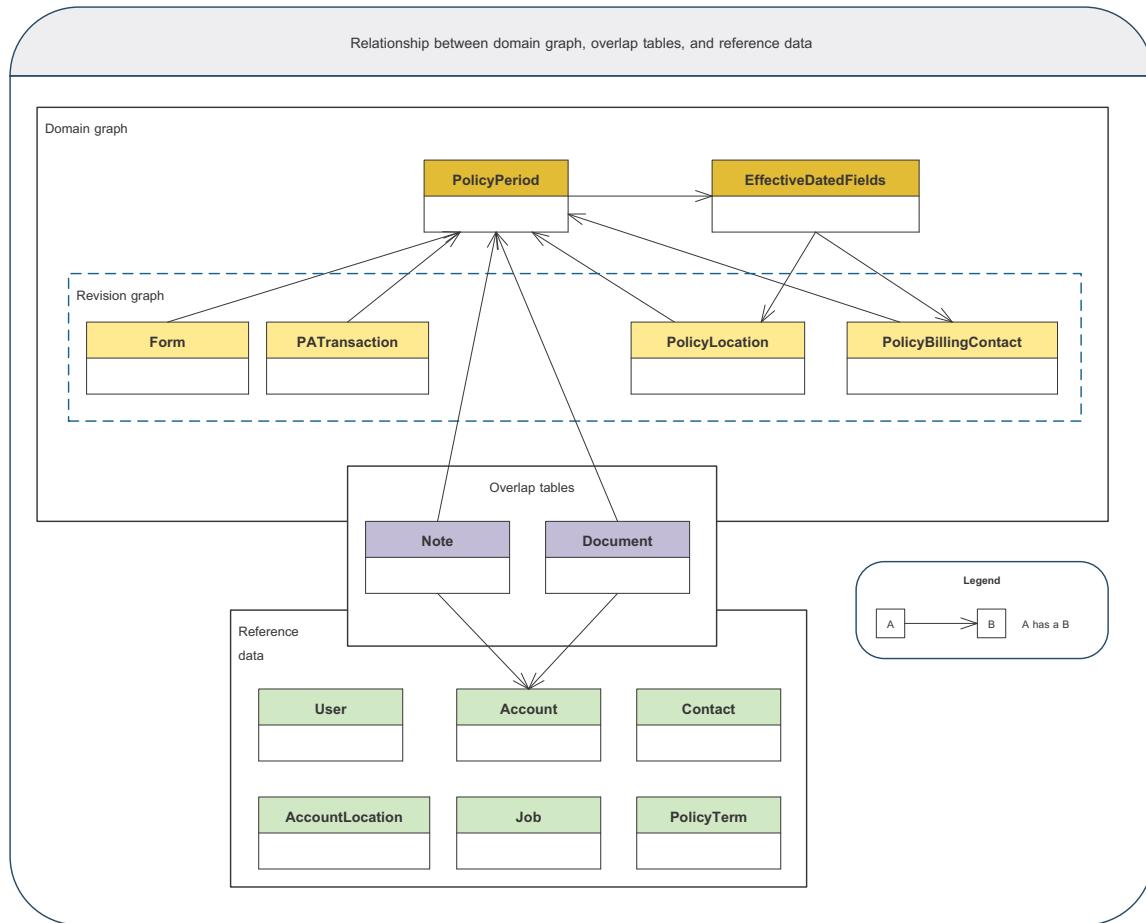
Any new entity type that you define and want to add to the domain graph must correctly define a foreign key to an entity type in the graph. This foreign key defines which object *owns* the other. There are several different types of ownership or ways to implement foreign keys between objects:

Ownership	Description
Object ownership	<p>A child object (B) has a foreign key to its parent object (A). In general, a foreign key to object A from object B means that A owns B. The direction of a solid arrow between two objects in the following illustration of the graph points to the owner.</p> <p>In PolicyCenter, the foreign key is defined through the effective dated branch as described in “Ownership Through the Effective Dated Branch” on page 249.</p>
Inverse ownership	<p>A parent object (A) has a foreign key to a child object (B). Thus, the direction of the foreign key for ownership is the inverse of typical object ownership. For inverse ownership, set the owner attribute on the foreign key defined on object A to true. The direction of a dashed arrow between two objects in the following illustration of the graph points to the owner.</p> <p>IMPORTANT Guidewire discourages the use of inverse ownership relationships. The PolicyCenter data model supports inverse ownership relationships for the rare case in which upgrading the database is unduly cumbersome or time consuming. As a general rule, do not use this type of relationship.</p>

The following illustration, which shows portions of the entities in the PolicyCenter data model, shows the relationships between entities and foreign keys. The illustration contains entities in the domain graph and in reference data, and the Note and Document entities, which are overlap tables.

The PolicyPeriod entity implements the RootInfo delegate. The PolicyPeriod and EffectiveDatedFields entities implement the Extractable delegate through inheritance. However, PolicyCenter does not remove the PolicyPeriod and EffectiveDatedFields entities from the database during archiving.

During archiving, PolicyCenter removes the entities in the revision graph from the database. PolicyCenter removes entities in the overlap tables if they apply to the policy period.



In the PolicyCenter data model:

- The **PolicyPeriod** object—and only that object—implements the **RootInfo** delegate.
- The **EffectiveDatedFields** object is part of the revision graph and the domain graph. It is retained when archiving occurs.
- All objects in the domain graph implement the **Extractable** delegate.
- The **Note** object implements a number of delegates, including the **Extractable** delegate. Objects that implement the **OverlapTable** delegate can exist in the domain graph and as reference data. However, individual rows in the table can exist only in one table or the other.

Domain Graph Validation

The PolicyCenter server performs a series of tests on the domain graph during startup. For some of these tests, failure prevents the server from starting. For other tests, failure allows the server to start with warnings. These tests uncover potential problems with the domain graph, dependent on business logic.

Graph Validation Errors That Prevent the Server From Starting

The following table describes validation tests of the domain graph that prevent the server from starting if any test fails.

Test	Description
Domain graph not partitioned	Verifies that all domain graph tables are reachable through the root entities.
Edge tables in domain graph have required foreign keys	Verifies that if edgeTable is set on an entity in the domain graph, it must have all of the following: <ul style="list-style-type: none"> An owned foreign key back to one of its parents An unowned foreign key to that same parent.
No cycles in domain graph	Looks for circular references in the domain graph. The domain graph is the set of tables and their relationships that define an aggregate of associated objects. PolicyCenter treats these objects as a single unit for purposes of data changes. <p>Circular references in the graph cause issues for this process.</p> <p>The domain graph cannot have any cycle in its <i>is owned by</i> relationships. Thus, the following example fails validation: A <i>is owned by</i> B <i>is owned by</i> C is owned by A.</p> <p>You need to resolve any cycles by sorting out the ownership relationships.</p> <p>The server reports the error and prints the graph problem in DOT format. You can use the DOT format output to view the graph visually by using graph visualization software, such as GraphViz.</p>
Domain graph entities implement Extractable	Ensures that all entities inside the domain graph implement the Extractable delegate. This test also verifies that no entities outside of the domain graph implement the Extractable delegate.
Overlap tables implement OverlapTable	Verifies that all overlap tables implement the OverlapTable delegate. An overlap table contains rows that can exist either in the domain graph or as part of reference data, but not both. Overlap tables must implement the following delegates: <ul style="list-style-type: none"> Extractable OverlapTable
Entities in domain graph keyable	Verifies that all entities in the domain graph are keyable. This requirement enables you to reference the entity by ID.
Reference entities retireable	Verifies that all reference entities in the domain graph points are retireable.
Exceptions to links from outside the domain graph are outside the domain graph	Verifies that exceptions to links from outside the domain graph are actually from entities that are outside of the domain graph.

See also

- “Viewing the Visual Domain Graph” on page 250

Graph Validation Warnings That Let the Server Start

The following table describes validation tests of the domain graph that let the server start, regardless of failure. PolicyCenter lets the server start even upon failure of these tests because business logic may prevent the erroneous situation. After the server starts, you can view the warnings from these tests on the **Domain Graph Info** screen.

Test	Description
Nothing outside the domain graph points to the domain graph	There must not be foreign keys from entities outside of the domain graph to entities in the domain graph. This prevents foreign key violations as PolicyCenter traverses the domain graph.
Null links cannot make node unreachable	PolicyCenter constructs the domain graph by looking at foreign keys. However, this can create a disconnected graph if a nullable foreign key is null. This test is a warning rather than one that prevents the server from starting as it is possible to use business logic to prevent the issue.

See also

- To learn how to view graph validation warnings, see “Viewing the Textual Domain Graph” on page 250

Working with Changes to the Data Model

Sometimes changes that you make to the data model inadvertently affect archiving.

An entity becomes part of the domain graph

In this case:

1. If a graph never referred to that entity, it does not appear in the XML. There is no issue.
2. If a graph that has been archived referred to an instance of that entity, it appears as a referenced entity in the XML. On retrieve, the archive process looks for that entity in the database and links to it. If you re-archive the graph, the archive process correctly exports the entity instance in the XML. There is no issue as long you do not delete the entity.

An entity is removed from the domain graph

In this case:

1. If the graph never referred to that entity, it does not appear in the XML. There is no issue.
2. It is possible that an entity instance was archived already. It is possible that someone or PolicyCenter retrieved the instance at a later time. On retrieve, the archived instance causes a duplicate key violation when the archive process attempts to insert the already archived instance into the database. In this case, you must turn the archived instance into a referenced entity.
 - a. Search for the duplicate instance in the database.
 - b. If found, call the `gw.api.archiving.upgrade.IArchivedEntity` method to create a new `IArchivedEntity` that is a referenced entity. The reference entity must exist in the database. You can create a new referenced entity of any type.

Working with Shared Entity Data

PolicyCenter does not permit an entity instance to exist in more than one instance of the domain graph. Existence in more than one instance of the domain graph violates the boundary of a unit of work. However, there are cases in which you might want to share entity instance data across multiple policies and their individual instances of the domain graph.

For example, you cannot have two policy periods that own the same Note instance. If you want to share an object, such as a Note, across multiple policy periods, extend the `PolicyPeriod` entity type and add a foreign key to the shared entity table. If you do so, there are several considerations of which you need to be aware:

- You cannot create foreign keys out of that shared entity to anything in the domain graph.
- Any code that you construct around the shared entity must be aware of archiving.

For example, the code must be aware that not all the related policy periods that the code references can exist in the main database.

If you intend to implement this type of solution, Guidewire strongly recommends that you consult with Guidewire Services on the project.

Eliminating Cycles From the Domain Graph

Two types of cycles can cause issues in the PolicyCenter data model and the domain graph:

- **Circular foreign key references** – Cycles that involve circular references between objects through the use of foreign keys. The concern with this type of cycle is the safe ordering of foreign keys between objects upon bundle commit.
- **Ownership cycles** – Cycles that involve the ownership of objects in the domain graph. The concern with this type of cycle is the ownership, both overt and implicit, of one object by another in the domain graph.

Circular Foreign Key References

A chain of foreign keys can form a cycle, also known as a *circular reference*. As an example of a circular reference, object A has a foreign key to object B, and B has a foreign key to A. Circular references can occur with more extensive chains of foreign keys, such as A refers to B, which refers to C, which refers to A.

The PolicyCenter data model and the domain graph do not permit circular references. Given a bundle that contains a circular reference between objects A and B, PolicyCenter cannot determine which object to commit first. For example, object A references new object B, which has not been committed yet. Thus, committing A before B is committed and present in the database causes a constraint violation.

Edge foreign keys provide the ability to avoid circular references in the data model. An edge foreign key from A to B introduces a new, hidden associative entity with a foreign key to A and a foreign key to B. The edge foreign key associates A and B without foreign keys directly between them. So, PolicyCenter can safely commit object A first, then new object B, and finally the hidden edge foreign key object.

See also

- “<edgeForeignKey>” on page 187

Ownership Cycles

A chain of ownership relationships can form a cycle known as an *ownership cycle* in the domain graph. Ownership cycles are hard to detect because ownership can flow either to or from an object that has a foreign key to another object.

By default, ownership flows in the same direction as foreign keys. For example, if B has a foreign key to A, B is owned by A. Sometimes it is necessary to invert the flow of ownership, so a foreign key points from the owner to the owned object instead.

Guidewire strongly recommends against the use edge foreign keys to resolve ownership cycles in the domain graph. Introduce edge foreign keys into the domain graph only to resolve circular foreign key references that require edge foreign keys for safe ordering during bundle commit.

See also

- “Ownership in the Domain Graph Through Foreign Keys” on page 248
- “Inverse Ownership in the Domain Graph” on page 249

Field Validation

This topic describes field validators in the PolicyCenter data model and how you can extend them.

This topic includes:

- “Field Validators” on page 259
- “Field Validator Definitions” on page 260
- “Modifying Field Validators” on page 263

See also

- “Configuring National Field Validation” on page 165 in the *Globalization Guide*

Field Validators

Field validators handle simple validation for a single field. A validator definition defines a *regular expression*, which a data field must match to be valid. It can also define an optional *input mask* that provides a visual indication to the user of the data to enter in the field.

Each field in PolicyCenter has a default validation based on its data type. For example, integer fields can contain only numbers. However, it is possible to use a field validator definition to override this default validation.

- You can apply field validators to simple data types, but not to typelists.
- You can modify field validators for existing fields, or create new validators for new fields.

For complex validation between fields, use validation-specific Gosu code instead of simple field validators.

Specifying the Properties of a Specific Field

Field validators specify only the validation properties for a general kind of input (for example, any postal code). They do not specify the properties of a specific field in a particular data view. Instead, detail views and editable list views include additional validation attributes in their configuration files.

Specifying Field Validators on a Delegate Entity

Apply any field validators for elements existing on a delegate entity to the delegate entity. Do not apply any field validators to the entities that inherit the elements from the delegate. This ensures that PolicyCenter applies the field validator uniformly to that data element in whatever code utilizes the delegate.

Field Validator Definitions

PolicyCenter stores the default field validator definitions in `fieldvalidators.xml`. This file contains a list of validator specifications for individual fields within PolicyCenter. Studio stores this file in the **Data Model Extensions** folder of the **Resources** tree. File `fieldvalidators.xml` contains the following sections:

XML element	Description
<code><FieldValidators></code>	Top XML element for the <code>fieldvalidators.xml</code> file.
<code><ValidatorDef></code>	Subelement that defines all of the validators. Each validator must have a unique name by which you can reference it.

Using the `fieldvalidators.xml` file, you can do the following:

- You can modify existing validators. For example, it is common for each installation site to represent policy numbers differently. You can define field validation to reflect these changes.
- You can add new validators for existing fields or custom extension fields.

The following XML example illustrates the structure of the base `fieldvalidators.xml` file:

```

<FieldValidators>
  <ValidatorDef name="Phone"
    description="Validator.Phone"
    input-mask="# #-###-### x###"
    value="[0-9]{3}-[0-9]{3}-[0-9]{4}([x][0-9]{0,4})?" />
  <ValidatorDef name="SSN"
    description="Validator.SSN"
    input-mask=""
    value="[0-9]{3}-[0-9]{2}-[0-9]{4}|[0-9]{2}-[0-9]{7}?" />
  ...
</FieldValidators>

```

Value Versus Input Mask

It is important to understand the difference between `value` and `input-mask`.

<code>value</code>	A value is a regular expression, which the field value must match in order for the data to be valid. PolicyCenter persists this value to the database, including any defined delimiters or characters other than the # character.
<code>input-mask</code>	An <code>input-mask</code> , which is optional, can assist the user in entering valid data. PolicyCenter displays the input mask to the user during editing or entering data into the field. For example, a # character indicates that the user can only enter a digit for this character. PolicyCenter interprets all other characters literally and includes them in the data.

After the user enters a value, PolicyCenter uses the regular expression to validate it. Typically, the input mask must lead to valid sequences for the regular expression or this can prevent the user from entering a valid value.

Guidewire PolicyCenter checks that the field data matches the field validator format (the regular expression) as it sets the field on the object. Thus, you cannot, for example, assign a value to a `ClaimNumber` field that does not match the acceptable `ClaimNumber` format as defined for this field in `fieldvalidators.xml`.

<FieldValidators>

The <FieldValidators> element is the root element in the `fieldvalidators.xml` file. It contains the following XML subelement.

<ValidatorDef>

The <ValidatorDef> element is the beginning element for the definition of a validator. This element has the following attributes:

- Name
- Value
- Description
- Input-Mask
- Format
- Placeholder-Char
- Floor, Ceiling

The following sections describe these attributes.

Name

The `name` attribute specifies the name of the validator. A field definition uses this attribute to specify which validator applies to the field.

Value

The `value` attribute specifies the acceptable values for the field. It is in the form of a regular expression. PolicyCenter does not persist this value (the regular expression definition) to the database.

Use regular expressions with `String` values only. Use floor and ceiling range values for numeric fields, for example, `Money`.

PolicyCenter uses the Apache library described in the following location for regular expression parsing:

<http://jakarta.apache.org/oro/api/org/apache/oro/text/regex/package-summary.html>

The following list describes some of the more useful items:

-
- () Parentheses define the order in which PolicyCenter evaluates an expression, just as with any parentheses.
 - [] Brackets indicate acceptable values. For example:
 - [Mm] indicates the letters M or m.
 - [0-9] indicates any value from 0 to 9.
 - [0-9a-zA-Z] indicates any alphanumeric character.
 - { } Braces indicate the number of characters. For example:
 - [0-9]{5} allows five positions containing any character (number) between 0 and 9.
 - {x} repeats the preceding value x times. For example, [0-9]{3} indicates any 3-digit integer such as 031 or 909, but not 16.
 - {x,y} indicates the preceding value can repeat between x and y times. For example, [abc]{1,3} allows values such as cab, b, or aa, but not rs or abca.
 - ? A question mark indicates one or zero occurrences of the preceding value. For example, [0-9]x? allows 3x or 3 but not 3xx. ([Mm][Pp][Hh])? means mph, MpH, MPH, or nothing.
 - ()? Values within parentheses followed by a question mark are optional. For example, (-[0-9]{4})? means that you can optionally have four more digits between 0 and 9 after a dash -.
 - * An asterisk means zero or more of the preceding value. For example, (abc)* means abc or abcabc but not ab.
-

-
- + A *plus* sign means one or more of the preceding value. For example, [0-9]+ means any number of integers between 0 and 9 (but none is not an option).
 - . A *period* is a wildcard character. For example:
 - .* means anything.
 - .+ means anything but the empty string.
 - ... means any string with three characters.
-

Description

The `description` attribute specifies the validation message to show to a user who enters bad input. The `description` refers to a key within the `display.properties` file that contains the actual description text. The naming convention for this display key is `Validator.validator_name`.

In the display text in the properties file, {0} represents the name of the field in question. PolicyCenter determines this at runtime dynamically.

Input-Mask

The `input-mask` attribute specifies an optional definition that provides a visual indication of what characters the user can enter. PolicyCenter displays the input mask to the user during data entry. It consists of the # symbol and other characters:

- The # symbol represents any character the user can type.
- Any other character represents itself in a non-editable form. For example, in an input mask of ###-##-##, the two hyphen characters are a non-editable part of the input field.
- Any empty input mask of "" is the same as not having the attribute at all.
- A special case is a mask with fixed characters on the end. PolicyCenter displays those characters outside of the text field. For example ####mph appears as a field #### with mph on the outside end of it.

Format

The `format` attribute works in a similar manner to the `input-mask` attribute. However, it is not currently in use.

Placeholder-Char

The `placeholder-char` attribute specifies a replacement value for the input mask display character, which defaults to a period (.). For example, use the `placeholder-char` attribute to display a dash character instead of the default period.

Floor, Ceiling

The `floor` and `ceiling` attributes are optional attributes that specify the minimum (`floor`) and maximum (`ceiling`) values for the field. For example, you can limit the range to 100-200 by setting `floor="100"` and `ceiling="200"`.

Use floor and ceiling range values for numeric fields only. For example, use the floor and ceiling attributes to define a Money validator:

```
<ValidatorDef description="Validator.Money"
               input-mask="" name="Money"
               ceiling="9999999999999999.99"
               floor="-9999999999999999.99"
               value=".*/>
```

Modifying Field Validators

Studio stores the `fieldvalidators.xml` file in the **Data Model Extensions** folder of the **Resources** tree. If you open this file for editing, Studio creates a custom copy of this file for you to edit, which PolicyCenter merges with the base configuration file at application runtime. Within your custom copy of `fieldvalidators` file, you can make a number of changes to the field validators.

You can, for example:

- Create a new field validator
- Modify attributes of an existing validator

The following code illustrates the syntax for these types of changes:

Create a new validator

```
<!-- Create a new validator -->
<ValidatorDef name="ExampleValidator" value="[A-z]{1,5}" description="Validator.Example"
    input-mask="#####"/>
```

Modify an existing validator definition

```
<!-- Modify a validator definition. Adding a ValidatorDef element with the same name as one defined
    in the base fieldvalidators.xml file replaces the base validator. -->
<ValidatorDef name="PolicyNumber" value="[0-9]{3}-[0-9]{5}" description="Validator.PolicyNumber"
    input-mask="###-#####"/>
```

Using `<columnOverride>` to Modify Field Validation

You use the `<columnOverride>` element in an extension file to override attributes on a field validator or to add a field validator to a field that does not contain one.

Adding a Field Validator to a Field

Occasionally, you want—or need—to add a validator to an application field that currently does not have one. You need to use a `<columnOverride>` element in the specific entity extension file. Use the following syntax:

```
<extension entityName="SomeEntity">
    <columnOverride name="SomeColumn">
        <columnParam name="validator" value="SomeCustomValidator"/>
    </columnOverride>
</extension>
```

Suppose that you want to create a validator for a Date of Birth field (`Person.DateOfBirth`). To create this validator, you need to perform the following steps in Studio.

1. Create a `Person.etx` file if one does not exist and add the following to it.

```
<extension entityName="Person">
    <columnOverride name="DateOfBirth">
        <columnParam name="validator" value="DateOfBirth"/>
    </columnOverride>
</extension>
```

2. Add a validation definition for the `DateOfBirth` validator to `fieldvalidators.xml`. For example:

```
<ValidatorDef description="Validator.DateOfBirth" ... name="DateOfBirth" .../>
```

In this case, you can potentially create different `DateOfBirth` validators in different country-specific `fieldvalidators` files.

Changing the Length of a Text Field

You can also use the `<columnOverride>` element to change the size (length) of the text that a user can enter into a text box or field. Guidewire makes a distinction between the `size` attribute and the `logicalSize` attribute.

- The `size` attribute is the length of the database column (if a `VARCHAR` column).
- The `logicalSize` attribute is the maximum length of the field that the application permits. It must not be greater than `size` attribute (if applicable).

In this case, you set the `logicalSize` parameter, not a `size` parameter. This parameter does not change the column length of the field in the database. You use the `logicalSize` parameter simply to set the field length in the PolicyCenter interface. For example:

```
<column-override name="EmailAddressHome">
  <columnParam name="logicalSize" value="42"/>
</column-override>
```

The use of the `logicalSize` parameter does not affect the actual length of the column in the database. It merely affects how many characters a user can enter into a text field.

Working with Typelists

This topic discusses typelists. Within Guidewire PolicyCenter, a *typelist* represents a predefined set of possible values, with each separate value defined as a *typecode*. Typically, you experience a typelist as drop-down list within Guidewire PolicyCenter that presents the set of available choices. You define and manage typelists through Guidewire Studio.

This topic includes:

- “What is a Typelist?” on page 266
- “Terms Related to Typelists” on page 266
- “Typelists and Typecodes” on page 266
- “Typelist Definition Files” on page 267
- “Different Kinds of Typelists” on page 268
- “Working with Typelists in Studio” on page 269
- “Typekey Fields” on page 272
- “Removing or Retiring a Typekey” on page 274
- “Typelist Filters” on page 275
- “Static Filters” on page 276
- “Dynamic Filters” on page 280
- “Dynamic Filters” on page 280
- “Typecode References in Gosu” on page 283
- “Mapping Typecodes to External System Codes” on page 284

See also

- “Localizing Typecodes” on page 47 in the *Globalization Guide*

What is a Typelist?

IMPORTANT Ensure that you fully understand the dependencies between typelists and other application files before you modify a typelist. Incorrect changes to a typelist can cause damage to the PolicyCenter data model.

Guidewire PolicyCenter displays many fields in the interface as drop-down lists of possible values. Guidewire calls the list of available values for a drop-down field a *typelist*. Typelists limit the acceptable values for many fields within the application. Thus, a typelist represents a predefined set of possible values, with each separate value defined as a *typecode*. Whenever there is a drop-down list in the PolicyCenter interface, it is usually a typelist.

For example, the PolicyCenter **Contact** page that you access as you edit policy vehicle information contains several different typelists (drop-down lists). One of these is the **Marital Status** typelist that provides the available values from which you can choose as you enter information about a vehicle owner.

Typelists are very common for coding fields on the root objects of an application. They are also common for status fields used for application logic. Some typelist usage examples from the *Data Dictionary* include:

- **Policy.ProductType** uses a simple list.
- **ExposureUnity.BodyType** uses a list filtered by **VehicleType** (that is, choices for this body type depend on the value of the vehicle type).

Besides displaying the text describing the different options in a drop-down list, typelists also serve a very important role in integration. Guidewire recommends that you design your typelists so that you can map their typecodes (values) to the set of codes used in your legacy applications. This is a very important step in making sure that you code a policy in PolicyCenter to values that can be understood by other applications within your company.

Terms Related to Typelists

There are several terms related to customizing drop-down lists within PolicyCenter. Since they sound quite similar, it is easy to confuse the meaning of each term. The following is a quick definition list for you to refer back to at any time for clarification purposes:

Term	Definition
Typelist	A defined set of values that are usually shown in a drop-down list within PolicyCenter.
Typecode	A specific value in a typelist.
Typefilter	A typelist that contains a static (fixed) set of values.
Keyfilter	A typelist that dynamically filters another typelist.
Typekey	The identifier for a field in the data model that represents a direct value chosen from an associated typelist.

Typelists and Typecodes

Within Guidewire PolicyCenter, a *typelist* represents a predefined set of possible values, with each separate value defined as a *typecode*. If Guidewire defines a typelist as **final**, it is not possible to add or delete typecodes from the typelist.

Internal Typecodes

Some typelists contain required internal typecodes that PolicyCenter references directly. Therefore, they must exist. Studio displays internal typecodes in gray, non-editable cells. This makes it impossible for you to edit or delete an internal typecode.

Localized Typecodes

It is possible to localize the individual typecodes in a typelist. See “Localizing Typecodes” on page 47 in the *Globalization Guide* for more information.

Mapping Typecodes to External System Codes

See the following:

- “Mapping Typecodes to External System Codes” on page 284
- “Mapping Typecodes to External System Codes” on page 93 in the *Integration Guide*

Typelist Definition Files

Similar to entity definitions, Guidewire PolicyCenter stores typelist definitions in XML files. There are three types of typelist files:

File type	Contains...
tti	A single typelist declaration. The name of the file prior to the extension corresponds to the name of the typelist. This can be either a Guidewire base configuration typelist or a custom typelist that you create through Studio.
txx	A single typelist extension. This can be a Guidewire-exposed base application extension or a custom typelist extension that you create.
tix	A single typelist extension for use by Guidewire only. These are generally Guidewire internal extensions to base application typelists, for use by a specific Guidewire application.

Always create, modify, and manage typelist definition files through PolicyCenter Studio. Guidewire specifically does not recommend or support manipulating the XML typelist files directly.

See also

- “Data Entity Metadata Files” on page 151

Different Kinds of Typelists

PolicyCenter organizes typelists into the following categories:

Category	Description
Internal	<p>Typelists that Guidewire controls as PolicyCenter requires these typelists for proper application operation. PolicyCenter depends on these lists for internal application logic. Guidewire designates internal typelists as <i>final</i> (meaning non-extendable). Thus, Guidewire restricts your ability to modify them.</p> <p>You can, however, override the following attribute values on these types of typelists:</p> <ul style="list-style-type: none"> • name • description • priority • retired
Extendable	<p>Typelists that you can customize. These typelists come with a set of example typecodes, but it is possible to modify these typecodes and to add your own typecodes. In some cases, these extendable typelists have internal typecode values that must exist for PolicyCenter to function properly. You cannot remove the internal typecodes, but you can modify any of the example typecodes.</p> <p>PolicyCenter designates internal typecodes by placing their code values in gray, non-editable cells. This makes these values inaccessible, and thus, impossible to modify.</p>
Custom Typelists	<p>Typelists that you add for specific purposes, for example, to work with a new custom field. These typelists are not part of the Guidewire base configuration. Studio automatically makes all custom typelists non-final (meaning extendable).</p>

Internal Typelists

A few of the typelists in the application are internal. Guidewire controls these typelists as PolicyCenter needs to know the list of acceptable values in advance to support application logic. Guidewire makes these typelists final by setting the `final` attribute to `true` in the data model. For example, `ActivityType` is an internal list because PolicyCenter implements specific behavior for known activity types.

Studio indicates internal typelists by shading the typelist icon light gray in the **Resources** tree. Studio also disables your ability to add additional typecodes to internal typelists.

The following are examples of internal typelists that you cannot change:

- `ActivityType`
- `CancellationTarget`
- `ActivityStatus`
- `AgencyStatus`
- `BillingTimePeriod`
- `FormSource`
- `PolicyRevisionStatus`

In some cases, Studio displays a typelist with a grayed-out icon in the **Resources** tree. This occurs if PolicyCenter manages the typelist (as opposed to the typelist being managed through an externally exposed XML file). In many cases, internally managed typelists are also internal typelists and explicitly have a `final` attribute set to `true`, which means that you cannot extend that typelist. There are, however, some typelists to which you can add additional typecodes (and are therefore not final), but, which PolicyCenter manages internally.

Overriding Attributes on Internal Typelists

While you cannot change an internal typelist, you can override the following attributes on an internal typelist:

- `name`
- `description`
- `priority`
- `retired`

Studio does not permit you to add additional categories (typecodes) to an internal typelist. You can, however, create a filter for the typelist.

To override a modifiable typelist attribute, first open the typelist in Guidewire Studio by selecting it from **Typelists** in the **Resources** tree. Then, select the typecode cell that applies and enter the desired data. You cannot change the typecode itself, only the attributes associated with the typecode.

Extendable Typelists

Many of the existing typelists are under your control. You cannot delete them or make them empty, but you can adjust the values (typecodes) within the list to meet your needs. PolicyCenter includes default typelists with sample typecodes in them. You can customize these typelists for your business needs by adding additional typecodes, if you want.

The **ActivityCategory** typelist is an example of an extendable typelist. If you want, you can add additional typecodes other than the sample values that Guidewire provides in the base configuration.

Custom Typelists

If you add a new field to the application, then it is possible that you also need to add an associated typelist. You can only access these typelists through new extension fields. For more information on how to add a new field to the data model, see “Extending a Base Configuration Entity” on page 215.

To create a custom typelist, in the **Project** window, navigate to **configuration** → **config** → **Extensions** → **Typelist**. Right-click on **Typelist**, and then click **New** → **Typelist**. Enter a name for the typelist, and then define your typecodes. PolicyCenter limits the number of characters in a typecode to 50 or less.

Working with Typelists in Studio

You create, manage and modify typelists within PolicyCenter using Guidewire Studio:

- To work with an existing extendable typelist, expand the **Typelist** folder in the Studio **Project** window and select the typelist from the list of existing typelists. This opens its editor in which you can change non-internal values or define new typecodes and filters.
- To view the values set for an internal typelist, select the typelist in the **Typelist** editor.
- To create a new custom typelist, navigate to **configuration** → **config** → **Extensions** → **Typelist**. Right-click on **Typelist**, and then click **New** → **Typelist**. Enter its name, and then define typecodes and filters for the typelist.

You cannot add a new typecode to, or modify an existing typecode of, a final typelist. However, it is possible to create filters for the typelist that modify its behavior within Guidewire PolicyCenter.

The Typelists Editor

If you modify an existing typelist, ensure that you thoroughly understand which other typelists depend on the typecode values in the typelist being modified. You must also update any related typelists as well. For example, any modifications that you make to the **Coverage** typelist can potentially affect the **CoverageType** typelist that the **Coverage** typelist filters. Therefore, you must update all of the related typelists as well.

After you select a typelist from the **Typelist** folder, Studio opens a typelist editor showing configuration options for that typelist.

The Studio Typelists Editor Interface

The top portion of the **Typelist** editor contains the following fields:

- **Description**

- Table name
- Final

The Description Field

PolicyCenter transfers the value that you enter in the **New → Typelist** dialog for the type list name to the **Description** field in the typelist editor. It is possible to edit this field.

Guidewire recommends that you add a `_Ext` suffix to the value that you enter for the type list name. This ensures that the name of any typelist that you create does not conflict with a Guidewire typelist implemented in a future database upgrade.

The Table Name Field

By default, Guidewire uses `pctl_typelist-name` as the name of the typelist table. However, if you want a different table name, you can override the default value by specifying a value in the **Table name** field for that typelist in Studio. If you override the default value, the table name becomes `pctl_table-name`.

Guidewire restricts the typelist table name to ASCII letters, digits, and underscore. Guidewire also places limits on the length of the name. However, if you choose, you can override the name of the typelist, which, in turn, overrides the table name stored in the database.

Thus:

- If you do not provide a value for the **Table name** field, then PolicyCenter uses the **Name** value and limits the table name to a maximum of 25 characters.
- If you do provide a value for the **Table name** field, then this overrides the value that you set in the **Name** field. However, the maximum table name length is still 25 characters.

Field	Value entered in...	Maximum length	Database table name
Name	New Typelist dialog	25 characters	<code>pctl_typelist-name</code>
Table name	Typelists editor	25 characters	<code>pctl_table-name</code>

The Final Field

A **final** typelist is a typelist to which you cannot add additional typecodes. You can, however, override the **name**, **description**, **priority**, and **retired** attributes. Studio marks typelists defined as final with a grayed-out icon. All custom typelists that you create are non-final.

The Studio Typelists Editor Tabs

The **Typelist** editor screen contains a number of tabs. Some of these tabs are not visible until you make a selection in the **Codes** tab. Each tab provides different functionality.

Tab	Use to...	See...
Codes	Enter a typecode and set its attributes	• “Entering Typecodes” on page 271
Filters	Define a fixed subset of a typelist to use as a static filter.	• “Static Filters” on page 276
Categories	Create a typelist filter that depends on the typecodes in a different typelist. This is a subtab. You must select a typecode to see this tab.	• “Dynamic Filters” on page 280

For information on how to localize typecodes using the Studio **Typelist Localization** editor, see “Localizing Typecodes” on page 47 in the *Globalization Guide*.

To create a new typelist

1. in the Project window, navigate to configuration → config → Extensions → Typelist.
2. Right-click on Typelist, and then click New → Typelist.
3. Enter the typelist name in the New Typelist dialog. PolicyCenter uses this name to uniquely identify this typelist in the data model.
4. Enter a description. Use the **Description** field to create a longer text description to identify how PolicyCenter uses this typelist. This text appears in places like the *Data Dictionary*.
5. Verify that the (Boolean) **Final** field is set to **false**. Studio automatically sets this field to false for any typelist that you create. You have no control over this setting. This field has the following meanings:

True	You cannot add or delete typecodes from the typelist. You can only override certain attribute fields.
False	You can modify or delete typecodes from this typelist, except for typecodes designated as internal, which you cannot delete. (You cannot remove internal typecodes, but you can modify their name, description, and other fields.)

Entering Typecodes

You use the **Codes** tab to enter typecodes for this typelist and to set various attributes for the typecodes. Each typecode represents one value in the drop-down list. Every typelist must have at least one typecode. Within this tab, you can set the following:

Field	Description
Code	A unique ID for internal Guidewire use. Enter a string containing only letters, digits, or the following characters: <ul style="list-style-type: none">• a dot (.)• a colon (:) Do not include white space or use a hyphen (-). Use this code to map to your legacy systems for import and export of PolicyCenter data. The code must be unique within the list. PolicyCenter limits the number of characters in a typecode to 50 or less. See also “Mapping Typecodes to External System Codes” on page 284.
Name	The text that is visible within PolicyCenter in the drop-down lists within the application. You can use white space and longer descriptions. However, limit the number of characters to an amount that does not cause the drop-down list to be too wide on the screen. The maximum name size is 256 characters.
Description	A longer description of this typecode. The maximum description size is 512 characters. PolicyCenter displays the text in this field in the <i>PolicyCenter Data Dictionary</i> .
Priority	A value that determines the sort order of the typecodes (lowest priority first, by default). You use this to sort the codes within the drop-down list and to sort a list of activities, for example, by priority. If you omit this value, PolicyCenter sorts the list alphabetically by name. If desired, you can specify priorities for some typecodes but not others. This causes PolicyCenter to order the prioritized ones at the top of the list with the unprioritized ones alphabetized afterwards.
Retired	A Boolean flag that indicates that a typecode is no longer in use. It is still a valid value, but not offered as a choice in the drop-down list as a new value. PolicyCenter does not make changes to any existing objects that reference this typecode. If you do not enter a value, PolicyCenter assumes the value is false (the default value).

Naming New Typecodes

Guidewire recommends that you add a `_Ext` suffix to the **Code** value for any new typecodes that you create. Do this only if the **Code** value is legal on any external system that needs to use the value. If that value is not legal, then omit the `_Ext` suffix.

Maximum Typelist Size

Guidewire strongly recommends that you limit the maximum number of typecodes in a typelist to 250 items. Any number larger than that can cause performance issues. If you need more typecodes than the 250 limit, then use a lookup (reference) table and a query to generate the typelist. In any case, Guidewire does not support the use of more than 8000 typecodes on a typelist.

Typelists and the Data Model

Guidewire recommends that you regenerate the *Data Dictionary* after you add or modify a typelist. Guidewire does not require that you do this. However, regenerating the *Data Dictionary* is an excellent way to identify any flaws with your new or modified typelist.

During application start up, Guidewire upgrades the application database if there are any changes to the data model, which includes any changes to a typelist or typecode. (In actual practice, this only occurs if the `autoupgrade` option is set to `true` in `config.xml`, which is almost always the case.)

See also

- “Typelists and Typecodes” on page 266
- “Mapping Typecodes to External System Codes” on page 284
- “Mapping Typecodes to External System Codes” on page 93 in the *Integration Guide*

Typekey Fields

A *typekey field* is an entity field that PolicyCenter associates with a specific typelist in the user interface. The typelist determines the values that are possible for that field. Thus, the specified typelist limits the available field values to those defined in the typelist. (Or, if you filter the typelist, the field displays a subset of the typelist values.)

For a PolicyCenter field to use a typelist to set values requires the following:

1. The typelist must exist. If it does not exist, then you must create it using the **Typelist** editor in PolicyCenter Studio.
2. The typelist must exist as a `<typekey>` element on the entity that you use to populate the field. If the `<typekey>` element does not exist, then you must extend the entity and manually add the typekey.
3. The PCF file that defines the screen that contains your typelist field must reference the entity that you use to populate the field.

The following example illustrates how to use the **Priority** typelist to set the priority of an activity that you create in PolicyCenter.

Step 1: Define the Typelist in Studio

It is possible to set a priority on an activity, a value that indicates the priority of this activity with respect to other activities. In the base configuration, the **Priority** typelist includes the following typecodes:

- High
- Low
- Normal
- Urgent

You define both the **Priority** typelist and its typecodes (its valid values) through PolicyCenter Studio, through the **Typelists** editor. For information on using the **Typelists** editor, see “Working with Typelists in Studio” on page 269.

Step 2: Add Typekeys to the Entity Definition File

For an entity to be able to access and use a typelist, you need to define a `<typekey>` element on that entity. You use the `<typekey>` element to specify the typelist in the entity metadata.

For example, in the base configuration, Guidewire declares a number of `<typekey>` elements on the `Activity` entity (`Activity.eti`), including the `Priority` typekey:

```
<entity entity="Activity" ... >
  ...
  <typekey default="task"
    desc="The class of the activity."
    name="ActivityClass"
    nullok="false"
    typelist="ActivityClass"/>
  <typekey desc="Priority of the activity with respect to other activities."
    name="Priority"
    nullok="false"
    typelist="Priority"/>
  <typekey default="open"
    desc="Status of the activity."
    exportable="false"
    name="Status"
    nullok="false"
    typelist="ActivityStatus"/>
  <typekey default="general"
    desc="Type of the activity."
    name="Type"
    nullok="false"
    typelist="ActivityType"/>
  <typekey desc="Validation level that this object passed (if any) before it was stored."
    exportable="false"
    name="ValidationLevel"
    typelist="ValidationLevel"/>
  ...
</entity>
```

Notice that the `<typekey>` element uses the following syntax:

```
<typekey desc="DescriptionString" name="FieldName" typelist="Typelist" />
```

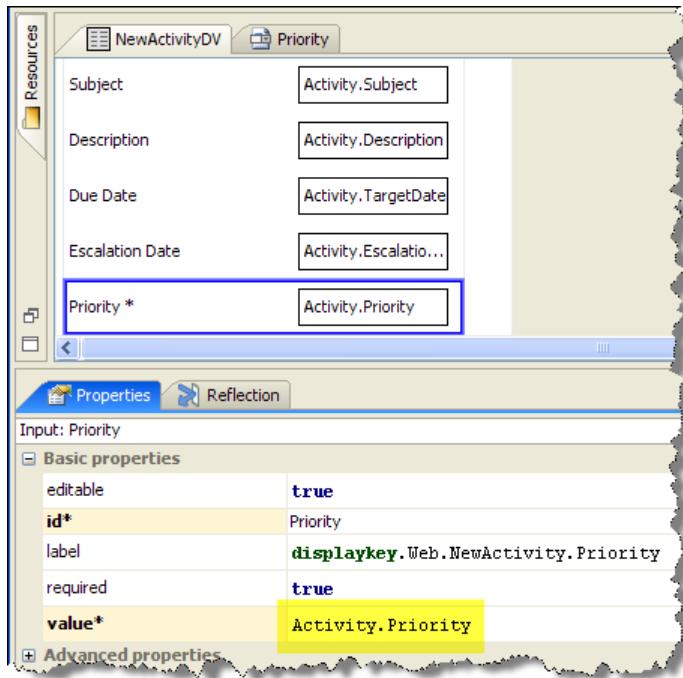
See also

- For information on the `<typekey>` element, see “`<typekey>`” on page 199.
- For information on how to create data model entities, see “The PolicyCenter Data Model” on page 149.
- For information on how to modify existing data model entities, see “Modifying the Base Data Model” on page 211.

Step 3: Reference the Typelist in the PCF File

Within Guidewire PolicyCenter, you can create a new activity. As you do so, you set a number of fields, including the priority for that activity. In order for PolicyCenter to render a `Priority` field on the screen, it must exist in the PCF file that PolicyCenter uses to render the screen.

Thus, the PolicyCenter **NewActivityDV** PCF file contains a **Priority** field with a value of `Activity.Priority`.



See also

- For information on working with the PCF editor, see “Using the PCF Editor” on page 289.
- For information on working with PCF files in general, see “Introduction to Page Configuration” on page 303.

Step 4: Update the Product Model

Guidewire recommends that you regenerate the *PolicyCenter Data Dictionary* before proceeding. If you have made any mistakes in the previous steps, regenerating the data dictionary helps to identify those mistakes.

In any case, you need to stop and restart the application server before you can view your changes in the PolicyCenter interface. Restarting the application server forces PolicyCenter to upgrade the data model in the application database.

Removing or Retiring a Typekey

Ensure that you fully understand the dependencies between typelists and other application files before you modify a typelist.

In general, Guidewire does not recommend that you make changes to existing typelists other than the following:

- Extending a non-final typelist to add additional typekeys.
- Retiring a typekey, which makes it invisible in the PolicyCenter interface, but leaves the typekey in the application database.

Be very careful of removing typekeys from a typelist as it is possible that multiple application files reference that particular typekey. Removing a typekey incorrectly can cause the application server to not start. Guidewire recommends that you retire a typekey rather than remove it.

It is not possible to remove a typekey from a typelist marked as final. It is also not possible to remove a typekey marked as internal. PolicyCenter indicates internal typekeys by placing their typecode values in gray, non-editable cells. This makes these typecode values inaccessible, and thus, impossible to modify.

Removing a Typekey

Suppose that you delete the `email_sent` typekey from the base configuration `DocumentType` typelist for some reason. If you remove this typekey, then you must also update all others part of the application install and disallow the production of documents of that type. In particular, you must remove references to the typekey from any `.descriptor` file that references that typekey. In this case, a search of the document template files finds that the `CreateEmailSent.gosu.htm.descriptor` file references `email_sent`.

To remove a typekey

1. Navigate to the typelist that contains the typekey that you want to retire.
2. Click the typekey, and then click **Remove** .
3. Search for additional references to the typekey in the application files and remove any that apply. Pay particular attention to `.descriptor` files. To remove a typekey reference:
 - a. Perform a case-insensitive text search throughout the application files to find all references to the deleted typekey.
 - b. Open these files in Studio and modify as necessary.

To retire a typekey

1. Navigate to the typelist that contains the typekey that you want to retire.
2. Select the **Retired** cell of the typekey that you want to retire.
3. Set the cell value to **true**.

If you retire a typekey, Guidewire recommends that you perform the steps outlined in *To remove a typekey* to identify any issues with the retirement:

- Verify all Studio resources.
- Perform a case-insensitive search in the application files for the retired typekey.

Typelist Filters

It is possible to configure a typelist so that PolicyCenter filters the typelist values so that they do not all appear in the drop-down list (typelist) in the PolicyCenter interface. Guidewire divides typelist filters into the following categories:

Type	Creates...	See...
Static	A fixed (static) subset of the values on a typelist. You can create filters that: <ul style="list-style-type: none">• Include certain specific typecodes on the typelist only.• Include certain specific categories of typecodes on the typelist.• Exclude certain specific typecodes from the full list of the typecodes on the typelist	"Static Filters" on page 276
Dynamic	A dynamic subset of the values on a typelist. You can create filters that: <ul style="list-style-type: none">• Associate one or more typecodes on a parent typelist with one or more typecodes on a child typelist.• Associate all the typecodes on a parent typelist with one or more typecodes on a child typelist.	"Dynamic Filters" on page 280

Static Filters

A *static* typelist filter causes the typelist to display only a subset of the typecodes for that typelist. Therefore, a static filter narrows the list of typecodes to show in the typelist view in the application. Guidewire calls this kind of typelist filter a static *typefilter*.

You define a static filter at the level of the typelist. You do this through the Studio Typelists editor, by defining a filter on the **Filters** tab for that particular typelist.

Studio manages the typelist XML file for you automatically. If you examine this file, you see that Studio uses the following XML syntax to define a static typelist filter. (In this case, a static filter that defines—or includes—a subset of the available typecodes.)

```
<typelistextension xmlns="http://guidewire.com/typelists" desc="Yes, no or unknown" name="YesNo">
  <typecode code="No" desc="No" name="No" priority="2"/>
  <typecode code="Yes" desc="Yes" name="Yes" priority="1"/>
  <typecode code="Unknown" desc="Unknown" name="Unknown" priority="3"/>
  <typefilter desc="Only display Yes and No typelist values" name="YesNoOnly">
    <include code="Yes"/>
    <include code="No"/>
  </typefilter>
</typelistextension>
```

Notice that the XML declares each typecode on the typelist (Yes, No, and Unknown). It then specifies a filter named **YesNoOnly** that limits the available values to simply Yes and No. This is static (fixed) filter.

For more information on the **<typefilter>** element, see “**<typekey>**” on page 199.

To create a static filter

1. Define the typecodes for this typelist in the Studio Typelist editor. See “Working with Typelists in Studio” on page 269 for details.
2. Select the **Filters** tab on this typelist in the **Typelist** editor.
3. Click **Add** and enter the following information for your static filter:

Attribute	Description
Name	The name of the filter. PolicyCenter uses this value to determine if a field uses this filter.
Description	Description of the context for which to use this typefilter.
Include All?	(Boolean) Typically, you only set this value to true if you use the exclude functionality. <ul style="list-style-type: none"> • True indicates that the typelist view starts with the full list of typecodes. You then use exclusions to narrow down the list. • False (the default) instructs PolicyCenter to use values set in the various subpanes to modify the typelist view in the application.

4. Use the fields in the following panes on the **Filters** tab to create a fixed subset of the typecodes for use in the static filter.

Subpane	Use to...	See...
Categories	Specify one or more typecodes to include by category within the filtered typelist view.	“Creating a Static Filter Using Categories” on page 277
Includes	Specify one or more typecodes to include within the filtered typelist view.	“Creating a Static Filter Using Includes” on page 278
Excludes	Specify one or more typecodes to exclude from the full list of typecodes for this typelist.	“Creating a Static Filter Using Excludes” on page 279

5. In the appropriate data model file, add a <typefilter> element to the child <typekey> for this typelist. To be useful, you must declare a static typelist filter (a typefilter) on that entity. Use the following XML syntax:

```
<typekey name="FieldName" typelist="Typelist" desc="DescriptionString" typefilter="FilterName"/>
```

You must manually add a typelist to an entity definition file. Studio does not do this for you. For example:

- The following code adds an unfiltered YesNo typelist to an entity:

```
<typekey desc="Some Yes/No question." name="YesNoUnknown" typelist="YesNo"/>
```

- The following code adds a YesNoOnly filtered YesNo typelist to an entity:

```
<typekey desc="Some other yes or no question." name="YesNo" nullok="true" typefilter="YesNoOnly" typelist="YesNo"/>
```

See “Typekey Fields” on page 272 for more information on declaring a typelist on an entity.

6. (Optional) Regenerate the *Data Dictionary* and verify that there are no validation errors. Use the following command in the PolicyCenter application bin directory to regenerate the *Data Dictionary*:

```
gwpc regen-dictionary
```

7. Stop and restart the application server to update the data model.

Creating a Static Filter Using Categories

Suppose that you want to filter a list of United States cities by state. (Say that you want to only show a list of appropriate cities if you select a certain state.) To create this filter, you need to first to define a City typelist (if one does not exist). You then need to populate the typelist with a few sample cities:

City typecodes	Location
ABQ	Albuquerque, NM
ALB	Albany, NY
LA	Los Angeles, NM
NY	New York, NY
SF	San Francisco, CA
SND	San Diego, CA
SNF	Santa Fe, NM

Then, for each City typecode, you need to set a category, similar to the following. You do this by selecting each typecode in turn, then clicking Add in the Categories pane in the Codes tab and entering the appropriate information:

City typecode	Associated typelist	Associated typecode
ABQ	State	NM
ALB	State	NY
LA	State	CA
NY	State	NY
SF	State	CA
SND	State	CA
SNF	State	NM

To generalize this example to regions outside the United States, you could associate the Jurisdiction typelist and a specific jurisdiction with each city typecode instead.

After making your choices, you have something that looks similar to the following:

Code	Name	Description	Priority	Retired
ABQ	Albuquerque	Albuquerque, NM	-1	false
SNF	Santa Fe	Santa Fe, NM	-1	false
LA	Los Angeles	Los Angeles, CA	-1	false
SND	San Diego	San Diego, CA	-1	false
NY	New York	New York, NY	-1	false
ALB	Albany	Albany, NY	-1	false

TypeList	Code
State	NM

This neatly categorizes each typecode by state.

On the **Filters** tab, click **Add** and enter **NewMexico** for the filter name. Now, in the **Categories** pane (on the **Filters** tab), enter the following:

Filter name	TypeList	Code
NewMexico	State	NM

This action creates a static category filter that only contains cities that exist in the state of New Mexico. Initially, the typelist contains Albuquerque and Santa Fe. If you add additional cities to the list at a later time that also exist in New Mexico, then the typelist displays those cities as well.

To be useful, you need to also do the following:

- Add the typelist to the entity that you want to display the typelist in the PolicyCenter user interface.
- Reference the typelist in the PCF file in which you want to display the typelist.

See “Typekey Fields” on page 272 for more information on declaring a typelist on an entity and referencing that typelist in a PCF file. In general, though, you need to add something similar to the entity definition that want to display the typelist:

```
<typekey name="NewMexico" typelist="City" typefilter="NewMexico" nullok="true"/>
```

Creating a Static Filter Using Includes

Suppose that you want to create a filtered typelist that displays zone codes that are in use only in Canada and not any other country. One way to create the filter is to use an **Includes** filter on the **ZoneTypes** typelist.

In this example, you want the typelist to display only the following:

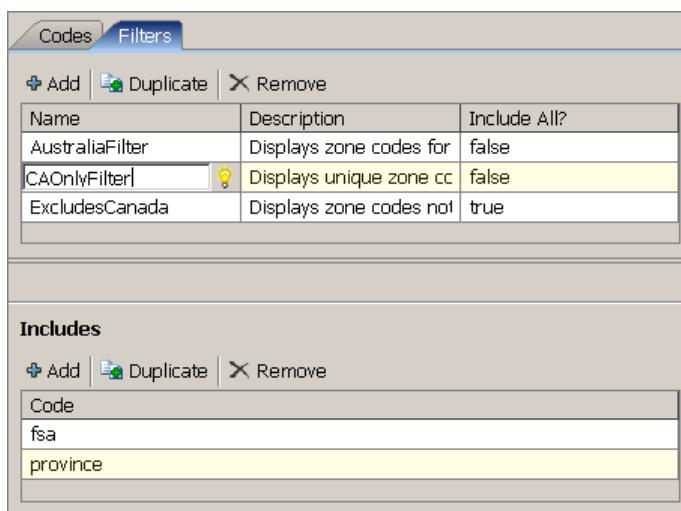
- fsa

- province

ZoneType typecode	Associated typelist	Associated typecode
city	Country	CA (Canada) US (United States)
county	Country	US
fsa	Country	CA
locality	Country	AU (Australia)
postcode	Country	AU
province	Country	CA
state	Country	AU US
zip	Country	US

To create an Include filter

1. Open the typelist that you want to filter in the Studio Typelists editor.
2. Navigate to the Filters tab.
3. Add the filter name to the list of filters. For example, call the filter that only displays certain zone type for the country of Canada CAOnlyFilter.
4. Finally, add the typecodes you want to include in the typelist in the Includes pane.



Creating a Static Filter Using Excludes

Suppose (for some reason) that you want to create a filtered typelist that displays all of the zone codes except those that are in use in Canada. You want to display the complete list of typecodes except for the following:

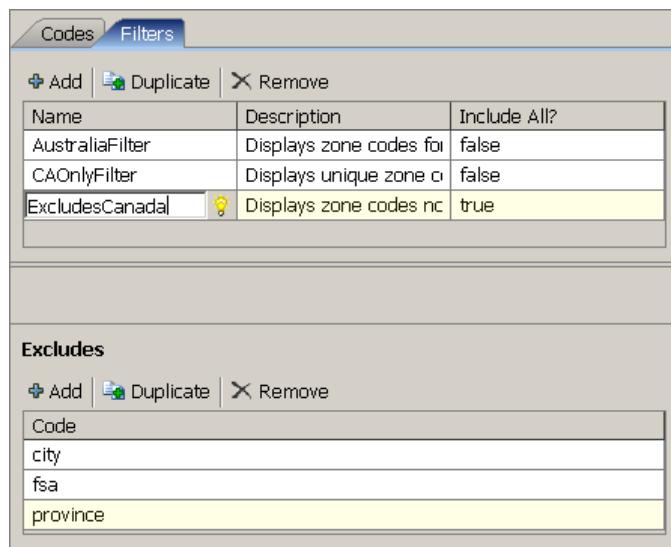
- city
- fsa
- province

ZoneType typecode	Associated typelist	Associated typecode
city	Country	CA (Canada) US (United States)
county	Country	US

ZoneType typecode	Associated typelist	Associated typecode
fsa	Country	CA
locality	Country	AU (Australia)
postcode	Country	AU
province	Country	CA
state	Country	AU US
zip	Country	US

To create an Excludes filter

1. Open the typelist that you want to filter in the Studio Typelists editor.
2. Navigate to the Filters tab.
3. Add the filter name to the list of filters. For example, call the filter that displays zone types that do not exist in Canada ExcludesCanada.
4. Finally, add the typecodes you want to exclude from the full set of typecodes for this typelist in the Excludes pane. Notice that you also set the **Include All?** value to true. This ensures that you start with a full set of typecodes.



Dynamic Filters

A *typecode filter* uses *categories* and *category lists* at the typecode level to restrict or filter a typelist. Typecode filters function in an equivalent manner to dependent filters in that the a parent typecode filters the available values on the child typecode.

You define a typecode filter directly on a typecode. You do this through the Studio Typelist editor, by defining a filter on the Codes tab for a particular typecode. To create this filter, you select a specific typecode and set a filter (category) on that typecode.

There are two types of typecode filters that you can define on the **Codes** tab:

Filter type	Use to...
Category	Associate one or more typecodes on a parent typelist with one or more typecodes on a child typelist.
Category list	Associate all the typecodes on a parent typelist with one or more typecodes on a child typelist.

Category Typecode Filters

- You use a *category* filter to associate one or more typecodes from one or more typelists with a specific typecode on the filtered typelist.
- You define a *category* filter in the **Typelist** editor on the **Codes** tab using the **Categories** pane.

Studio manages the typelist XML file for you automatically. If you examine this file, you see that Studio uses the following XML syntax to define a typecode category filter:

```
<typecode code="DependentTypecode" desc="DescriptionString" typelist="DependentTypeListName">
  <category code="Typecode1" typelist="Typelist1"/>
  <category code="Typecode2" typelist="Typelist1"/>
  <category code="Typecode3" typelist="Typelist2"/>
  ...
</typecode>
```

Category List Typecode Filters

- You use a *category list* filter to associate all of the typecodes from one or more typelists with a specific typecode on the filtered typelist.
- You define a *category list* filter in the **Typelists** editor on the **Codes** tab using the **Category Lists** pane.

Studio manages the typelist XML file for you automatically. If you examine this file, you see that Studio uses the following XML syntax to define a typecode category list filter:

```
<typecode code="Typecode" desc="DescriptionString" typelist="DependentTypeListName">
  <categorylist typelist="TypelistName"/>
</typecode>
```

Creating a Dynamic Filter

In general, to create a dynamic filter, you need to do the following:

- Step 1: Set the Category Filter on Each Typecode
- Step 2: Declare the Category Filter on an Entity
- Step 3: Set the PolicyCenter Field Value in the PCF File
- Step 4: Update the Product Model

As the process of declaring a typecode filter on an entity can be difficult to understand conceptually, it is simplest to proceed with an example. Within Guidewire PolicyCenter, a user with administrative privileges can define a new activity pattern (**Administration** → **Activity Patterns** → **New Activity Pattern**). Within the **New Activity Pattern** screen, you see several drop-down lists:

- Type
- Category

PolicyCenter automatically sets the value of **Type** to **General**. (You cannot edit this field as Guidewire sets the value of **editable** to **false** for this field in the base configuration.) This value determines the available choices that you see in the **Category** drop-down list. For example:

- Correspondence
- General
- Interview
- New mail
- ...

The **ActivityCategory** typelist is the typelist that controls what you see in the **Category** field in PolicyCenter. If you open this typelist in the Studio Typelists editor, you can choose each typecode in the list one after another. As you select each typecode in turn, notice that the Studio associates each typecode with a **Typelist** and a **Code** value in the **Categories** pane. (In this case, Studio associates each **ActivityCategory** typecode with an **ActivityType** typecode.) Thus, PolicyCenter filters each individual typecode in this typelist so that it is only available for selection if you first select the associated typelist and typecode.

Step 1: Set the Category Filter on Each Typecode

The process is the same to create a category list typecode filter. In that case, you associate a single typelist (and all its typecodes) with each individual typecode on the dependent typelist. You make the association by selecting a typecode in the dependent typelist and setting the controlling typelist in the **Category Lists** pane.

Open the **ActivityCategory** typelist and select each typecode in turn. As you do so, you see that Studio associates each typecode with an **ActivityType.Code** value in the **Categories** pane. For example, if you select the **interview** typecode, you see that Guidewire associates this typecode with an **ActivityType.Code** value of **general**. This is the process that you need to duplicate if you create a custom filtered typelist or if you customize an existing typelist. The following graphic illustrates this process.

The screenshot shows the Guidewire Studio Typelists editor interface. At the top, there is a header bar with tabs for 'Codes' and 'Filters'. Below this, there are three main sections:

- Codes:** A table listing typecodes with columns for Code, Name, Description, and Priority. The rows include approval, approvaldenied, correspondence, general, and interview. The 'interview' row is highlighted with a yellow background.
- Categories:** A table titled 'Categories' with columns for Typelist and Code. It shows a single entry where the Typelist is 'ActivityType' and the Code is 'general'. Both the 'ActivityType' and 'general' cells are highlighted with yellow backgrounds.
- Final:** A configuration section with a dropdown menu set to 'false'.

Step 2: Declare the Category Filter on an Entity

The question then becomes how do you set this behavior on the **ActivityPattern** entity. In other words, what XML code do you need to add to the **ActivityPattern** entity to enable the **ActivityType** typelist to control the values shown in the PolicyCenter **Category** field? The following code sample illustrates what you need to do. You must add a typekey for both the parent (**ActivityType**) typelist and the dependent child (**ActivityCategory**) typelist.

```
<entity xmlns="http://guidewire.com/datamodel" ... entity="ActivityPattern" ...>
...
<typekey default="general" desc="Type of the activity." name="Type" typelist="ActivityType"/>
...
<typekey ... name="Category" typelist="ActivityCategory">
  <keyfilters>
    <keyfilter name="Type"/>
  </keyfilters>
</typekey>
...
</entity>
```

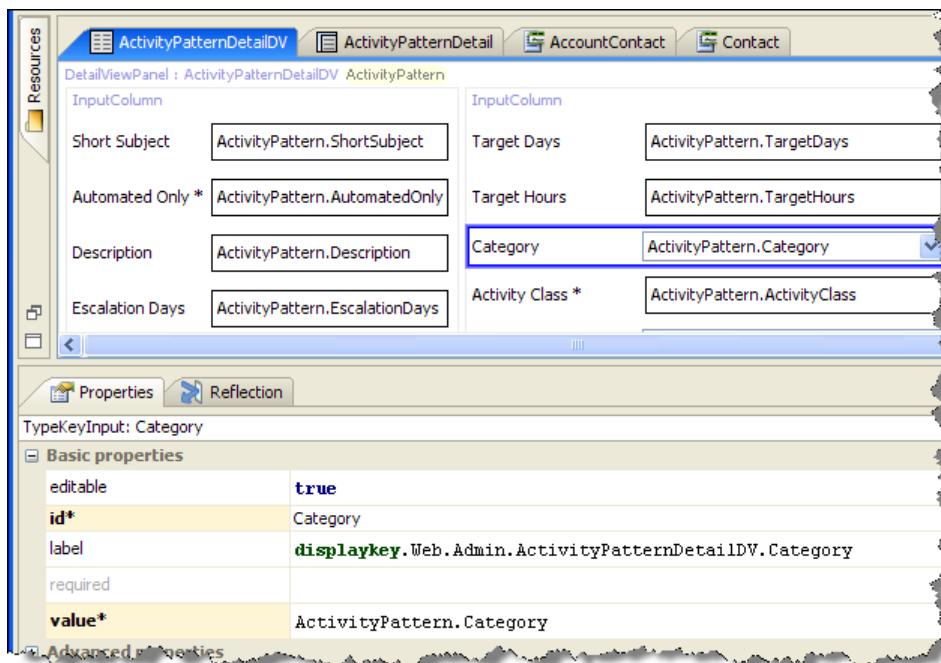
The sample code first defines a <typekey> element with name="Type" and typelist="ActivityType". This is the controlling (parent) typelist. The code then defines a second typelist (ActivityCategory) with a keyfilter name="Type". It is the typelist referenced by the <keyfilter> element that controls the behavior of the typelist named in the <typekey> element. Thus, the value of ActivityType.Code controls the associated typecode on the dependent ActivityCategory typelist.

For more information on the <keyfilter> element, see “<typekey>” on page 199.

Step 3: Set the PolicyCenter Field Value in the PCF File

After you declare these two typelists on the ActivityCategory entity, you need to link the typelists to the appropriate fields on the PolicyCenter New Activity Pattern screen. To access an entity typelist, you need to use the entity.TypeList syntax. For example, to access the ActivityCategory typelist on the ActivityPattern entity, use ActivityPattern.Category with Category being the name of the typelist.

You do this in the PolicyCenter ActivityPatternDetailDV.pcf file:



Step 4: Update the Product Model

Guidewire recommends that you regenerate the *PolicyCenter Data Dictionary* before proceeding. If you have made any mistakes in the previous steps, regenerating the data dictionary helps to identify those mistakes.

In any case, you need to stop and restart the application server before you can view your changes in the PolicyCenter interface. Restarting the application server forces PolicyCenter to upgrade the data model in the application database.

Typecode References in Gosu

To refer to a specific typecode in Gosu, use the following syntax.

`typekey.TypeList.TC_Typecode`

For example, the default State typelist has typecodes for states in the US and provinces in Canada.

`State`

`...`

Code	Name
IL	Illinois
...	

To refer to the typecode for the state of Illinois in the typelist State, use the following Gosu expression.

`typekey.STATE.TC_IL`

You must prefix the code in the object path expressions for typecodes with `TC_`.

Note: Use code completion in Studio to build complete object path expressions for typecodes. Type “`typekey.`” to begin, and work your way down to the typecode that you want.

Mapping Typecodes to External System Codes

Your PolicyCenter application can share or exchange data with one or more external applications. If you use this functionality, Guidewire recommends that you configure the PolicyCenter typelists to include typecode values that are one-to-one matches to those in the external applications. If the typecode values match, sending data to, or receiving data from, those applications requires no additional effort on the part of an integration development team.

However, there can be more complex cases in which mapping typecodes one-to-one is not feasible. For example, suppose that it is necessary to map multiple external applications to the same PolicyCenter typecode, but the external applications do not match. Alternatively, suppose that you extend your typecode schema in PolicyCenter. This can possibly cause a situation in which three different codes in PolicyCenter represent a single (less granular) code in the other application.

To handle these more complex cases, you need to edit resource file `typecodemapping.xml` within Guidewire Studio. (You can find this file in the `configuration → config → typelists.mapping` folder.) This file specifies a *namespace* for each external application. Then, you identify the individual unique typecode maps by typelist.

A Typecode Mapping Example

The following code sample illustrates a simple `typecodemapping.xml` file:

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

  <typelist name="AccountSegment">
    <mapping typecode="PR" namespace="accounting" alias="ACT" />
  </typelist>
</typecodemapping>
```

The `namespacelist` tag contains one or more `namespace` tags—one for each external application. Then, to map the actual codes, you specify one or more `typelist` tags as required. Each `typelist` tag refers to a single internal or external typelist in the application. The `typelist`, in turn, contains one or more `mapping` tags. Each `mapping` tag must contain the following attributes:

<code>typecode</code>	Specifies the PolicyCenter typecode.
<code>namespace</code>	Specifies the name space to which PolicyCenter maps the typecode
<code>alias</code>	Specifies the code in the external application.

In the previous example, the PR PolicyCenter code maps to an external application named `accounting`. You can create multiple mapping entries for the same PolicyCenter typecode or the same name space. For example, the following specifies a mapping between multiple PolicyCenter codes and a single external code:

```
<typelist name="BoatType">
  <mapping typecode="AI" namespace="accounting" alias="boat" />
```

```
<mapping typecode="HY" namespace="accounting" alias="boat" />
</typeList>
```

After you define the mappings, from an external system you can use the `TypeListToolsAPI` web service to translate the mappings. See the “Mapping Typecodes to External System Codes” on page 93 in the *Integration Guide*.

User Interface Configuration

Using the PCF Editor

This topic covers how to work with PCF (Page Configuration Format) files in Guidewire Studio.

This topic includes:

- “Page Configuration (PCF) Editor” on page 289
- “Page Canvas Overview” on page 290
- “Creating a New PCF File” on page 290
- “Working with Shared or Included Files” on page 291
- “Page Config Menu” on page 293
- “Toolbox Tab” on page 294
- “Structure Tab” on page 294
- “Properties Tab” on page 295
- “PCF Elements” on page 297
- “Working with Elements” on page 297

Page Configuration (PCF) Editor

Guidewire PolicyCenter uses *page configuration format* (PCF) files to render the PolicyCenter interface. You use the PCF editor in Studio to manage existing PCF files and create new ones.

The PCF editor provides the following features:

- Intelligent Gosu coding
- Instant feedback whenever a PCF file changes
- Drag-and-drop composition of PCF pages and their graphical elements
- High-level view of PCF page groupings
- Ability to localize the display keys used in a PCF page

The PCF editor comprises three areas:

- In the center pane, the graphical page *canvas*, which provides drag-and-drop capabilities for composing and managing the graphical and interactive elements on a page.
- In the right-hand pane, the following tabs:
 - **Toolbox** – Contains a search box and a list of elements that you can insert into the page
 - **Structure** – Shows the containment hierarchical of the elements on the page
- At the bottom, the **Properties** tabs at the bottom of the screen.

Page Canvas Overview

The center pane of the PCF editor provides the graphical page *canvas*. The page canvas provides drag-and-drop capabilities for composing and managing the graphical and interactive elements on a page.

The page canvas displays the following:

- Elements that represent page content, such inputs and similar items, in simplified versions to illustrate how they appear within the PolicyCenter user interface.
- Elements that function primarily as containers (data views, for example) as light gray boxes, with a header indicating the element type and ID.
- Elements that define or expose additional Gosu symbols to their descendants as light gray boxes, with a list of symbols at the top. If you move your mouse over a symbol, Studio shows a tooltip with the name, type, and initial value of the symbol.
 - If the symbol represents a Require, the tooltip indicates this as well.
 - If you click a symbol name, Studio selects the containing element, and then opens the appropriate properties tab for editing whatever is providing the symbol. Finally, if necessary, Studio selects the symbol in the **Properties** tab.
- Elements that are conditionally visible with a dotted border.
- Elements that iterate over a set of data and produce their contents once for each element in the data by a single copy of the contents. It follows this with an ellipsis to indicate iteration.
- RowIterator widgets with inferred header and footer cells in the position in which they appear within PolicyCenter.

Creating a New PCF File

Guidewire Studio displays PCF files in an organizational hierarchy. To create a new PCF file, you need to first decide its location in the PCF hierarchy. If the hierarchy does not contain a PCF folder at the organization level that suits your needs, first create one before you create your new PCF File.

PCF folder names are case-insensitive and must be unique within the PDF hierarchy. You cannot create a PCF folder name that differs from an existing PCF folder name by case only.

To create a new PCF folder

1. In the Project window, navigate to **configuration** → **config** → **Page Configuration**, and expand it.
2. Select a node one level above the level in which you need to create the new PCF folder (node).
3. Right-click and click **New** → **PCF folder**.
4. Enter the folder name in the **New Folder** dialog.

To create a new PCF file

1. In the Project window, navigate to **configuration** → **config** → **Page Configuration**, and expand it.

2. Select the node in which you want to create the new PCF file.
3. Right-click and click New → PCF file.
4. Enter the file name in the New PCF File dialog.
5. Select the PCF file type to create.
6. Enter a mode. (Any element that dynamically includes this widget must specify the same mode.) This field is only active with specific file types. See “Working with Shared or Included Files” on page 291 for more information.

The following table lists the file type icons.

Icon	File type	Icon	File type	Icon	File type	Icon	File type
	Page		Input Set		Navigation Tree		Toolbar Buttons
	Popup		List View		Panel Row		Wizard
	Card View		List-Detail View		Panel Set		Wizard Steps
	Chart View		Location Group		Popup Wizard		Wizard Step Subgroup
	Detail View		Menu Actions Set		Row Set		Worksheet
	Entry Point		Menu Items		Screen		
	Exit Point		Menu Links Set		Tab Bar		
	Info Bar		Navigation Forward		Template Page		

Working with Shared or Included Files

A *shared element* or *shared section* is any PCF element that has the following characteristics:

- The PCF element is not a top-level element, meaning it is not a Page, Popup, or Wizard, for example.
- The PCF element exists in its own file.

Guidewire calls this a shared section because it is possible to share the element (or file) between multiple top-level elements. PolicyCenter automatically propagates any changes that you make to the shared section to all other PCF elements that include the shared section.

You cannot select elements within the included file and the included elements do not display a highlight or a tooltip as you move the mouse cursor over it. For all intents and purposes, included elements are flat content of the element in the current file that includes them.

However, PolicyCenter displays a PCF element that includes the contents of another file or element with a blue overlay. This overlay is cumulative. Studio displays included elements that are several levels deep in a darker shade of blue. If you double-click an area with a blue overlay, Studio opens the included file in a new editor view.

Right-clicking anywhere on the canvas and toggling **Show included sections** or toggling **Show included sections** from the **Page Config** menu disables the representation of the included files. Studio displays the text of the reference expression instead.

Understanding PCF Modes

Certain included files or elements are *modal*. The basic idea is that you can define several different file versions, or modes, of a single shared section. Thus, any PCF page that includes the section can decide at run-time which file version to use, possibly based on the value of some variable.

If it is possible for the PCF file to have multiple modal versions, then you see **Shared section mode** above the shared area (which Studio shades or high-lights in blue). If you click the mode, Studio shows a drop-down of all the possible modes. You can use the drop-down to select a different modal file. If you do so, then Studio updates the screen to reflect your change.

For example, in ClaimCenter (the other Guidewire applications provide similar examples), PCF file `ExposureDetailScreen` contains a shared area. The mode drop-down contains a number of possible modal files that you can embed into the `ExposureDetailsScreen`. In this screen, the drop-down shows the following:

- Baggage
- Bodilyinjurydamage
- Content
- EmployerLiability
- ...

To determine if a PCF file has multiple modal versions, you can also look at the PCF file names in Studio. If you see multiple file names that include a common name followed by a dot then a different name, then this is a modal file. For example, in ClaimCenter, you see the following under the `exposures` node in the Resources tree:

- `ExposureDetailDV.Baggage`
- `ExposureDetailDV.Bodilyinjurydamage`
- `ExposureDetailDV.Content`
- `ExposureDetailDV.Employerliability`
- ...

Each individual file is a modal version of the `ExposureDetailDV` PCF file, which you can embed into another file, in this case, the `ExposureDetailsScreen`.

Setting a PCF Mode

It is only possible to set a *mode* on a PCF file as you create that PCF file. Selecting a file type that allows modes enables the **Mode** text field in the **New PCF File** dialog. Selecting a file type that does not allow modes disables the **Mode** text field.

You are able to set a mode with the following file types only:

- | | | |
|----------------------------|-------------------|------------------------|
| • Accelerated Menu Actions | • List View | • Row Set |
| • Card View | • Menu Action Set | • Screen |
| • Chart View | • Menu Items | • Toolbar Buttons |
| • Detail View | • Menu Links Set | • Wizard Steps |
| • Info Bar | • Modal Cell | • Wizard Step Subgroup |
| • Input Set | • Panel Set | |

Typically, you use a Gosu expression to define the mode for an included section. You can make this expression either a hard-coded string literal or you can use the expression to evaluate a variable or a method call. For example, PCF file `AddressPanelSet` (in PolicyCenter) uses `selectedAddress.CountryCode` as the mode expression variable.

A hard-coded string guarantees that the included section always uses the same mode regardless of the data on the page. If using a hard-coded string expression, Studio shows only that mode and does not show a drop-down above the blue area.

Creating New Modal PCF files

It is not possible to change or modify the mode of a base configuration PCF file. You can, however, use an existing modal file as a template to create a new (different) modal version of that file. To do this:

- Select the template file and duplicate it.
- Select the newly created file and change the mode of that file.

For example, suppose that you wanted to add a new modal version of the `ExposureDetailDV` PCF file, say `ExposureDetailDV.BusinessPropertydamage`. To do this:

1. Select the file that you intend to use as the template. For this example, select `ExposureDetailDV.Baggage`.
2. Right-click the template file and select **Duplicate**.
3. Enter the name of the new modal file in the **Duplicate PCF File** dialog and click **OK**. For this example, enter `ExposureDetailDV.BusinessPropertydamage`
Studio inserts the new file into the directory structure, colors the file name blue, and opens a view of the file automatically.
4. Modify the new modal file as required.

Include Files with Multiple Modes.

PolicyCenter provides the ability to use a single include file with multiple modes. In this way, you can re-use a single modal file in multiple PCF files. For example, in the base configuration, ClaimCenter defines PCF file `AddressBookAdditionalInfoInputSet.PersonVendor` with multiple modes:

- `PersonVendor`
- `Attorney`
- `Doctor`

To see this, open `AddressBookAdditionalInfoInputSet.PersonVendor` and select the entire file. (You see a solid blue line surrounding the file.) Examine the `mode` attribute in the **Properties** pane at the bottom of the screen. You see the following:

`PersonVendor|Attorney|Doctor`

To create an include file with multiple modes

1. Select the include file.
2. Right-click and select **Change mode....**
3. Enter the individual modes separated by a pipe symbol (|) in the **Change Mode** dialog.

Page Config Menu

If you open the PCF editor, Studio displays a **Page Config** menu on the main Studio menu bar. This menu contains a number of useful items.

Menu command	Use to...	See
<code>Change element type...</code>	Substitute a different element for the selected element. The dialog contains a list of element types that you can substitute for the selected element within the constraints of the PCF schema.	" Changing the Type of an Element " on page 299
<code>Edit comment...</code>	Attach a comment to any element on the canvas.	" Adding a Comment to an Element " on page 299
<code>Delete comment</code>	Remove a comment from an element.	" Adding a Comment to an Element " on page 299

Menu command	Use to...	See
Disable element	Disable an element by commenting out the widget. This prevents PolicyCenter from rendering the widget in the interface.	"Adding a Comment to an Element" on page 299
Enable element	Enable a previously disabled element. This action removes the surrounding comment tags from the element.	"Adding a Comment to an Element" on page 299
Link widgets	Link widgets on a parent page that spans multiple child PCF files. You use this particularly for explicit iterator references.	"Linking Widgets" on page 301
Show included sections	Toggle the visibility of child files embedded in a parent PCF file. If you disable the representation of the included files, Studio displays the text of the reference expression instead.	"Page Canvas Overview" on page 290
Find by ID	Find an element by its ID. The dialog contains a filter text field and a list of all elements on the canvas that have their id attribute set:	"Finding an Element on the Canvas" on page 300
Show element source	View the XML code for an element. Studio displays the XML code in a pop-up window.	"Viewing the Source of an Element" on page 300

Toolbox Tab

The **Toolbox** tab contains a search box and a list of widgets, divided into categories and subcategories.

- Clicking on a category name expands or collapses that category.
- Clicking on a subcategory name expands or collapses that subcategory as well.

Within the toolbox, Studio persists the state of each category (expanded or collapsed) across all PCF editor views. It also persists the state of each category to each new Studio session.

Studio only displays widget categories containing widgets that are valid and available for use in the current PCF file. If you hover the mouse cursor over a widget name in the list, then Studio displays a description of that widget in a tooltip.

Search Box

You use the search box to filter the full set of widgets. Typing in the search box temporarily expands all widget categories and highlights:

- Any widgets whose category name matches the typed text
- Any widgets whose name matches the typed text
- Any widgets whose actual name in the XML matches the typed text
- Any widgets whose description contains the typed text

Clicking the X icon by the search box clears text from the box and stops filtering the widget list. Keyboard shortcut ALT+/ gives focus to the search box.

Structure Tab

The **Structure** tab shows the hierarchical structure of the PCF file as a tree. Each node in the tree represents a PCF element. Any children of the node are children of that element:

- If you click an element that represents a concrete element on the canvas, Studio selects that element on the canvas.

- If you click on an element that does not represent a concrete element on the canvas, then Studio first selects the containing element on the canvas. It then selects the appropriate properties tab with which to edit the clicked element. Finally, if necessary, Studio selects the clicked element in the properties tab (at the bottom of the screen).

Properties Tab

The **Properties** tab (at the bottom of the screen) displays all attributes of the selected element. Studio divides the attribute workspace into **Basic** and **Advanced** sections. You can expand or collapse a workspace section by clicking the title of that section. Studio maintains the expanded or collapsed state of a section across all element selections and persists this state to new Studio sessions.

The workspace displays each attribute as a row in a table, with the attribute name in the left column and the value in the right column. Studio grays out the name if you have not set a value for that attribute (if the attribute value is nothing or is only a default value). Studio also grays out the attribute value if it is a default value. If the PCF schema requires an attribute, Studio displays the attribute name in bold font, with an asterisk, and with a different background color.

If you hover the mouse cursor over an attribute name, Studio displays a tooltip with the documentation for that attribute.

For each attribute:

- If the attribute takes a non-Gosu string value, Studio displays the value in a text field.
- If the attribute takes a non-Gosu Boolean value, Studio displays the value in a drop-down menu with two choices, `true` and `false`.
- If the attribute takes an enumeration value, Studio displays the value in a drop-down menu with a choice for each value of the enumeration, plus a `<none selected>` option.
- If the attribute takes a Gosu value, Studio displays the value in a single-line Gosu editor. Gosu editor commands that operate on multiple lines have no effect in a single-line editor. (For example, the `SmartFix Add uses statement` command does not work in a single-line editor.) Studio displays the single-line editor with a red background if it contains any errors, and a yellow background if it contains warnings but no errors.
- If the attribute requires a return type, Studio colors the value background red under the following circumstances:
 - If the entered expression does not evaluate to that type
 - If the entered statement does not return a value of that type
- If the attribute requires a Boolean return value, Studio displays a drop-down menu on the right side with two options, `true` and `false`. If you select one of these options, Studio sets the text of the editor to the appropriate value.

If the value editor for an attribute has focus, Studio displays the attribute name in a different background color and adds an X icon. If you click the X icon, Studio sets the value of the attribute to its default.

If you press `Enter` on the keyboard while editing a property, Studio moves the focus to the next property in the list.

Child Lists

Some of the Properties tabs contain a *child list*. A child list contains a list of the selected element's child elements of a certain type, and a properties list for the selected child element. You can perform a number of operations on a child list, using the following tool icons.

- If the child list represents a single child type, Studio adds a new child element. Studio selects the newly added child automatically. If the child list represents multiple child types, Studio opens a drop-down menu of available child types. If you select a child type from the drop-down, Studio adds a child of that type and selects it automatically.
- If you select a child and click the delete icon, Studio removes the selected child. Studio disables this action if there is no selected child.
- If you select a child and click the up icon, Studio moves the selected child above the previous child. Studio disables the up icon if you do not first select a child, or if there are no other children above your selected child.
- If you click the down icon, Studio moves the selected child below the next child. Studio disables the down icon if you do not first select a child, or if there are not other children below your selected child.

Additional Properties Tabs

Depending on the children of a selected element, Studio displays additional subtabs.

Additional tabs	Description
Axes	If an element can have DomainAxis and RangeAxis children, Studio displays the Axes properties tab. This tab contains a child list of the DomainAxis and RangeAxis children for the selected element. If you select a RangeAxis, Studio displays a child list for its Interval children.
Code	If an element can have a Code child, Studio displays the Code properties tab. This tab contains a Gosu editor for editing the contents of the Code child. The Code editor has access to all the top-level symbols in the PCF file (for example, any required variables). However, you cannot incorporate any uses statements, nor does the Gosu editor provide the SmartFix to add uses statements automatically.
Data Series	If an element can have DataSeries and DualAxisDataSeries children, Studio displays the Data Series properties tab. This tab contains a child list of the DataSeries and DualAxisDataSeries children for the selected element.
Entry Points	If an element can have LocationEntryPoint children, Studio displays the Entry Points properties tab. This tab contains a child list of the LocationEntryPoint children for the selected element.
Exposes	If a parent PCF page contains an iterator that controls a ListView element defined in a separate PCF file, then Studio displays the Exposes tab on the child PCF file. You use this tab to set the iterator to use for the ListView element.
Filter Options	If an element can have ToolbarFilterOption and ToolbarFilterOptionGroup children, Studio displays the Filter Options properties tab. This tab contains a child list of the ToolbarFilterOption and ToolbarFilterOptionGroup children for the selected element.
Next Conditions	If an element can have NextCondition children, Studio displays the Next Conditions properties tab. This tab contains a child list of the NextCondition children for the selected element.
Reflection	If an element can have a Reflect child, Studio displays a Reflection properties tab. The Reflection tab has a checkbox for Enable client reflection, which indicates whether that element has a Reflect child. If you enable client reflection, the Reflection tab also contains a properties list for the Reflect element and a child list for its ReflectCondition children.
Required Variables	If an elements can have Require children, Studio displays the Required Variables properties tab. This tab contains a child list of the Require children for the selected element.
Scope	If an element can have Scope children, Studio displays the Scope properties tab. This tab contains a child list of the Scope children for the selected element.
Sorting	If an element can have IteratorSort children, Studio displays the Sorting properties tab. This tab contains a child list of the IteratorSort elements for the selected element.

Additional tabs	Description
Toolbar Flags	If an element can have ToolbarFlag children, Studio displays the Toolbar Flags properties tab. This tab contains a child list of the ToolbarFlag children of the selected element.
Variables	If an element can have Variable children, Studio displays the Variables properties tab. This tab contains a child list of the Variable children for the selected element.

PCF Elements

Studio displays a down arrow icon to the right of a non-menu element that contains menu-item children.

- If you click the down arrow, Studio opens a pop-up containing the children of the element.
- If you click anywhere on the canvas outside the pop-up, Studio dismisses the pop-up.

Studio displays elements that contain a comment with a comment icon in the upper right-hand corner of the widget. It shows disabled elements (commented-out elements) in a faded-out manner

Studio displays elements that cause a verification error with either a red overlay or a thick red border. It displays elements that cause a verification warning (but not an error) with either a yellow overlay or a thick yellow border.

If you move your mouse over an element:

- Studio highlights the element with a light border.
- If the element has a comment, Studio displays the text of the comment in a tooltip.
- If the element does not have a comment, but does have its desc attribute set, Studio displays the value of the desc attribute in a tooltip.
- If the element has any errors or warnings, Studio displays these in a tooltip along with any comment or desc text.

PCF Elements and the Properties Tab

If you click an element on the canvas, Studio selects that element and highlights it in a thick border. This action also opens the Properties tab in the workspace area at the bottom of the screen, if it is not already visible.

- If the element has an error or warning that is attributable to one of its attributes, Studio highlights that attribute in the Properties tab.
- If the element contains child elements not shown on the canvas, Studio displays additional Properties tabs in the workspace area.
- If the element has no errors or warnings, but a non-visible child element does, Studio brings the appropriate Properties tab for that child element to the front. If necessary, Studio selects that child element in the Properties tab.
- If there are additional Properties tabs that do not apply to the selected element, Studio closes them.
- If the tab that was at the front before you selected the element is still visible, it remains at the front. Otherwise, Studio brings the Properties tab to the front.

Clicking in the canvas area outside the area representing the file being edited, or clicking Escape, de-selects the currently selected element and closes all open Properties tabs.

Working with Elements

Page configuration format files contain three basic types of elements:

- Physical elements (buttons and inputs, and similar items) that have a visual presence in a live application.
- Behavioral elements (iterator sorting and client reflection, and similar items) that exist only to specify behavior of other elements.

- Structural elements (panels, screens, and similar items) that do not represent a single element in the Web interface, but instead indicate some grouping or other structure.

After you create a new page, you can select page elements from the **Toolbox** tab for inclusion in the page. PolicyCenter does not permit you to insert elements that are invalid for that page or grouping. After adding an element to a page, you can change its type if needed, rather than removing it and starting again.

Guidewire strongly recommends that you label the widgets that you create with unique IDs. Otherwise, you may find it difficult to identify that widget later.

You can perform the following actions with PCF elements:

- Adding an Element to the Canvas
- Changing the Type of an Element
- Adding a Comment to an Element
- Finding an Element on the Canvas
- Viewing the Source of an Element
- Duplicating an Element
- Deleting an Element
- Copying an Element
- Cutting an Element
- Pasting an Element

Adding an Element to the Canvas

To add a widget, click its name in the **Toolbox** and hold the mouse cursor down. As you begin to drag the widget, Studio changes the mouse cursor so that it includes the icon for that widget. Studio places a green line on the canvas at every location on the canvas that it is possible to place the widget. Studio highlights the green line that is nearest on the canvas to the cursor. Studio also overlays in green the element containing the highlighted green line.

- If the widget is a menu item of some sort, Studio overlays in green those widgets that can accept the item as a menu item.
- If a green widget is closer to the cursor than any of the green lines, Studio overlays it with a brighter green.

If you click Esc, Studio cancels the action, returns the cursor to normal, and makes the green lines and overlays disappear.

If you click again (or end the dragging operation), Studio adds the new widget at the location of the highlighted green line. (Or, Studio adds the widget as a child of the highlighted widget.) Studio sets all attributes of the new widget to their default value.

After Studio adds the new widget to the canvas, the cursor returns to normal and the green lines and overlays disappear. Studio selects this new widget automatically.

Moving an Element on the Canvas

If you click on a widget on the canvas, Studio picks up (selects) the widget. As you drag the widget, Studio moves the widget from its current location to the new location. This makes no changes to the attributes or descendants of the widget.

You can also CTRL+drag a widget on the canvas. This time, however, as you place the widget, Studio creates a duplicate of the original widget (including all attributes and descendants) and places the cloned widget at the target location.

Changing the Type of an Element

If you right-click an element and select **Change element type**, Studio opens the **Change Element Type** dialog. You can also select the element and then select **Change element type** from the **Page Config** commands on the menu bar.

This dialog contains a list of element types that you can substitute for the selected element within the constraints of the PCF schema. If you then select a new element type and click **OK**, Studio replaces the selected element with an element of the new type. It also transfers all attribute values and descendants that are valid on the new type.

However:

- If it is possible to select a new element type that does not allow one or more attributes supported by the selected (existing) element. In this case, Studio displays a message that indicates which attributes it plans to discard.
- If it is possible to select an element type that can not contain one or more children of the selected widget. In this case, Studio displays a message that indicates which children it plans to discard.

If there are no valid element types to which you can change the selected element, Studio disables the **Change element type** command.

Adding a Comment to an Element

It is possible to attach a comment to any element on the canvas. Studio indicates an element has a comment by placing a yellow note icon in the comment's upper right corner. If you hover the mouse over that element, then Studio displays the comment in a tooltip.

Adding a Comment

If you do one of the following, Studio opens a modal dialog with a text field for the element's comment:

- Right-click an element and select **Edit comment**
- Select the element and then select **Edit comment** from the **Page Config** commands on the menu bar

If the element already has a comment, Studio pre-populates the text field with the contents of the comment.

Deleting a Comment

If you do one of the following, Studio deletes the comment for that element:

- Right-click an element and select **Delete comment**
- Select the element and then select **Delete comment** from the **Page Config** commands on the menu bar

However, if the element has no comment, Studio disables the **Delete comment** command.

Disabling (Commenting-out) an Element

Commenting out a widget effectively prevents PolicyCenter from rendering the widget in the interface. If you do any of the following, Studio disables the element by surrounding it and its descendants with comment tags in the XML:

- Right-click an enabled element and select **Disable element**.
- Select the element and click **CTRL+/-**.
- Select **Disable element** from the **Page Config** commands on the menu bar.

If the element or any of its descendants have comments, Studio informs you that it is deleting the comments and prompts you to confirm the disable operation.

It is also possible to set the **visible** attribute on the widget to **false** to prevent PolicyCenter from rendering the widget. In this case, however, the XML file retains the widget. It still exists server-side at run time, although PolicyCenter does not render it, and the widget does not post data. Thus, commenting out the widget can possibly give you a marginal performance increase. Also, disabling the widget prevents it from causing errors, for example, if the signature of some function called by one of its attributes changes.

As it is the use of XML comment tags that disable the widget, you cannot then add a comment to the widget to describe why you disabled it. If you would like to add an explanation associated with the widget (recommended), then use the `widget desc` attribute. Studio displays this text in the tooltip if you hover the mouse over the widget in the PCF editor. (This does not produce a yellow note icon, however.)

Enabling an Element

If you do one of the following, Studio enables the element by removing the surrounding comment tags:

- Right-click a disabled element and select **Enable element**.
- Select the element and click **CTRL+/.**
- Select **Enable element** from the **Page Config** commands on the menu bar.

Finding an Element on the Canvas

If you do one of the following, Studio opens a semi-modal dialog. This dialog contains a filter text field and a list of all elements on the canvas that have their `id` attribute set:

- Right-click in the canvas area and select **Find by ID**.
- Click **CTRL+F12**.
- Select **Find by ID** from the **Page Config** commands on the menu bar.

As you type in the text field, Studio filters the visible elements to those whose ID matches the typed text. Selecting an element from the list selects it on the canvas.

Viewing the Source of an Element

To view the XML representation of an element, select the widget, then do one of the following:

- Right-click, and then select **Show element source**.
- Select **Show element source** from the **Page Config** menu.

Studio opens a small text window and displays the XML code associated with the selected element.

Duplicating an Element

If you do one of the following, Studio creates a duplicate of the element immediately after the current element:

- Right-click an element and select **Duplicate**.
- Select a widget and click **CTRL+D**.
- Select **Duplicate** from the **Edit** commands on the menu bar.

This includes all attribute values and descendants. Studio selects the duplicate widget automatically.

If the PCF schema permits the target widget to occur one time only within the parent widget (for example, a Screen), then attempting to duplicate the widget has no effect.

Deleting an Element

If you do one of the following, Studio deletes the element from the canvas:

- Right-click an element and select **Delete**.
- Select a widget and click **Delete**.
- Select **Delete** from the **Edit** commands on the menu bar.
- Select the **Delete** icon from the menu bar.

You cannot delete the root element of a PCF.

Copying an Element

If you do one of the following, Studio copies an XML representation of that widget and its descendants to the clipboard:

- Right-click an element and select **Copy**.
- Select a widget and click CTRL+C.
- Select **Copy** from the **Edit** commands on the menu bar.
- Select the **Copy** icon from the menu bar.

Cutting an Element

If you do one of the following, Studio copies an XML representation of that widget and its descendants to the clipboard and deletes the widget:

- Right-click an element and select **Cut**.
- Select a widget and click CTRL+C.
- Select **Cut** from the **Edit** commands on the menu bar.
- Select the **Cut** icon from the menu bar.

You cannot cut (remove) the root element of a PCF file.

Pasting an Element

If the content of the clipboard is valid XML representing a PCF widget, you can paste the widget by doing one of the following:

- Right-click the canvas and select **Paste**.
- Click CTRL+V.
- Select **Paste** from the **Edit** commands on the menu bar.
- Select the **Paste** icon from the menu bar.

Linking Widgets

A common feature in PCF pages is a **ListView** element controlled by a **RowEditor** iterator. PolicyCenter renders the list view as a table with multiple rows, with the iterator populating the data in the table rows. Frequently, the user clicks a button to activate certain functionality within the list view, for example, adding or deleting rows in the table.

In many cases, the PCF file is a parent page that contains embedded child PCF files that contain individual **ListView** elements. If this is the case, then you need to link the button on the parent PCF file with the iterator used to populate the child PCF files. To do so, you use the **Link widgets** command on the **Page Config** menu. This menu item provides a visual tool to link widgets on a parent page that spans multiple child PCF files. You use this particularly for explicit iterator references.

To link two widgets

1. Select a widget, for example a **CheckedValuesToolbarButton** widget.
2. Do one of the following:
 - Select **Link widgets** from the **Page Config** menu.
 - Right-click and select **Link widgets** from the context menu.
 - Press CTRL+L.

Studio changes the look of the mouse cursor to cross-hairs. Studio also changes the color of the target widget to light green.

3. Click the widget to which you want to link. Studio links the two widgets.

Introduction to Page Configuration

This topic provides an introduction to the concepts and files involved in configuring the web pages of the PolicyCenter user interface.

This topic includes:

- “Page Configuration Files” on page 303
- “Page Configuration Elements” on page 303
- “Getting Started Configuring Pages” on page 309
- “Modifying Style and Theme Elements” on page 311

Page Configuration Files

The pages in the PolicyCenter user interface are defined by XML files stored within each installed instance of the application. To configure your PolicyCenter interface, use Guidewire Studio to open and edit these files. The page configuration files are named with the file extension `.pcf`, and are therefore often called *PCF files*.

IMPORTANT Because the Guidewire platform interprets PCF files based on a hierarchy, you can only edit PCF files in the configuration module. Studio manages this hierarchy automatically. However, if you choose to edit PCF files without Studio, be aware that editing the wrong version of one of these files can prevent the application from starting. Guidewire expressly does not support editing PCF files outside of Guidewire Studio.

Page Configuration Elements

This section discusses the following topics:

- What is a PCF Element?
- Types of PCF Elements
- Identifying PCF Elements in the User Interface

Edited Resource Files Reside *Only* in Configuration Module

The `PolicyCenter/modules/configuration` directory is the only place for user-edited resources. During PolicyCenter start-up, a checksum process verifies that no files have changed in any directory except for those in the `configuration` directory. If this process detects an invalid checksum, the application refuses to start. In this case, you need to overwrite any changes to all modules *except* for the `configuration` directory and try again.

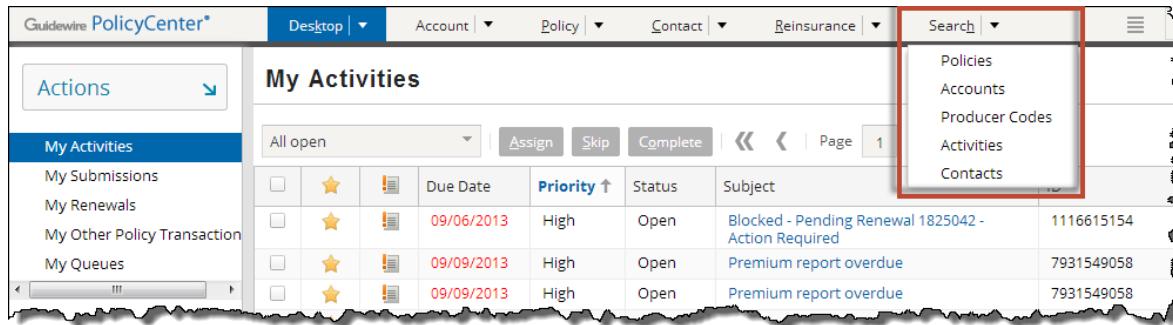
What is a PCF Element?

Guidewire defines each PCF file as a set of XML elements defined within the root `<PCF>` tag. Guidewire calls these XML elements *PCF elements*. These PCF elements define everything that you see in the PolicyCenter interface, as well as many things that you cannot see. For example, PCF elements include:

- Editors
- List views
- Detail views
- Buttons
- Popups
- Other PolicyCenter interface elements
- Non-visible objects that support the PolicyCenter interface elements, such as Gosu code that performs background actions after you click a button.

For a reference of all PCF elements and their attributes, see the *PCF Format Reference* in `PolicyCenter/modules/pcf.html` in your installation.

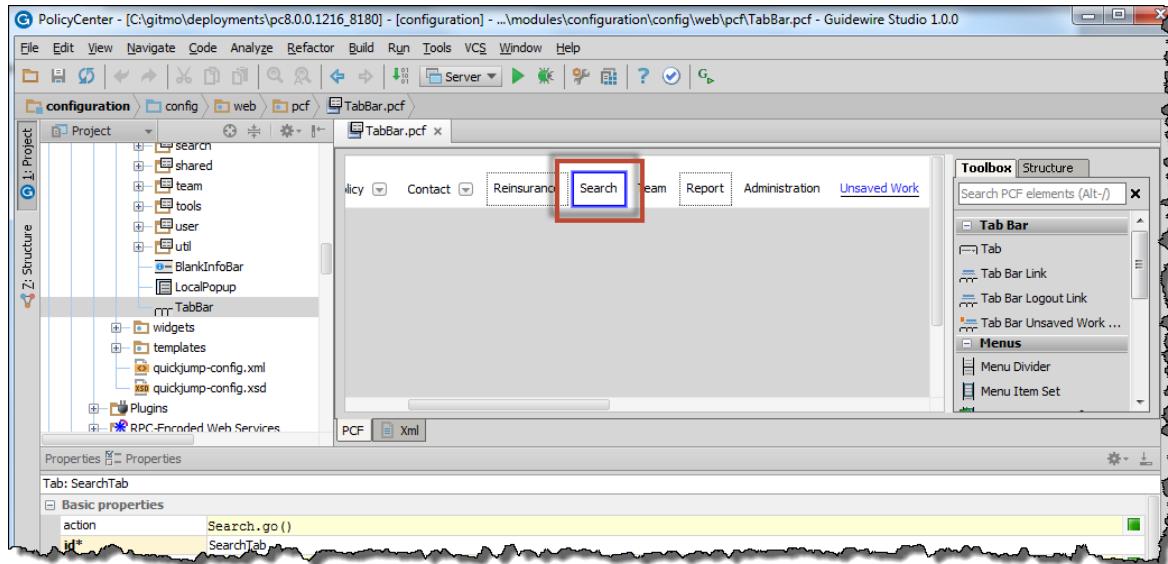
Every page in PolicyCenter uses multiple PCF elements. You define these elements separately, but PolicyCenter renders them together during page construction. For example, consider the tab bar available on most PolicyCenter pages:



The screenshot shows the Guidewire PolicyCenter interface with the title bar "Guidewire PolicyCenter". The main content area displays a "My Activities" grid with columns for Due Date, Priority, Status, and Subject. A search dropdown menu is open on the right, listing "Policies", "Accounts", "Producer Codes", "Activities", and "Contacts".

	Due Date	Priority	Status	Subject
09/06/2013	High	Open	Blocked - Pending Renewal 1825042 - Action Required	
09/09/2013	High	Open	Premium report overdue	
09/09/2013	High	Open	Premium report overdue	

Using **Ctrl+Shift+W**, you can discover that these elements are defined in the PCF file `TabBar.pcf`. In Guidewire Studio, you can open `TabBar.pcf` in the PCF Editor. Clicking on the arrow next to the Search tab in this file causes the search menu items to appear:



Types of PCF Elements

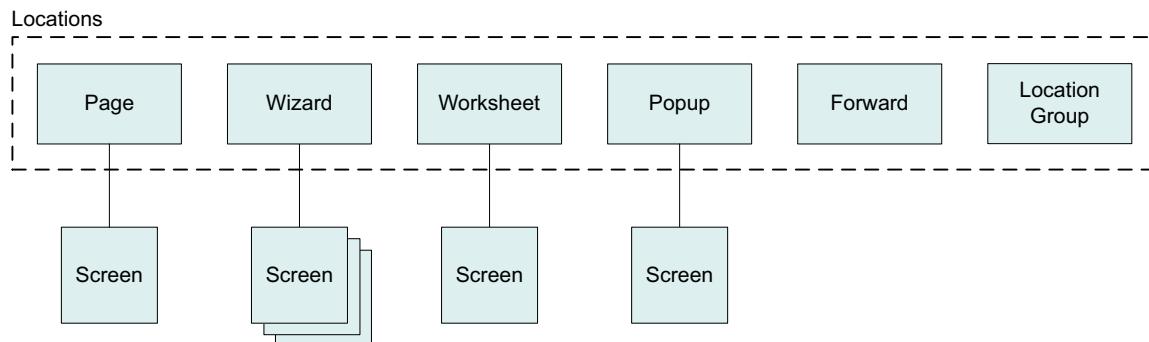
There are many kinds of PCF elements that you can define. These elements follow a hierarchical, container-based user interface model. To design them most effectively, you need understand the relationships between them thoroughly. Most PCF elements are of one of the following types:

- Locations
- Widgets

Locations

A *location* is a place to which you can navigate in the PolicyCenter interface. Locations are used primarily to provide a hierarchical organization of the interface elements, and to assist with navigation.

Locations include pages, wizards, worksheets, forwards, and location groups. Locations themselves do not define any visual content, but they can contain screens that do, as illustrated in the following diagram:



You can define the following types of locations:

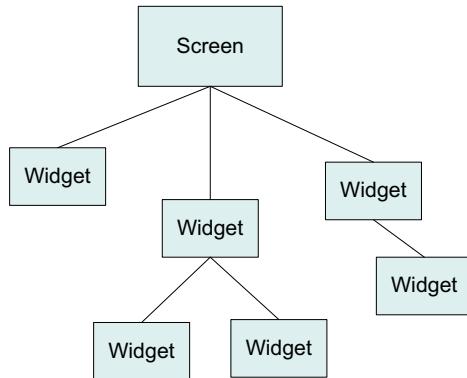
Location	Description
Page	A location with exactly one screen. The majority of locations defined in PolicyCenter are pages.
Wizard	A location with one or more screens, in which only one screen is active at a time. The contents of a wizard are usually not defined in PCF files, but are configured either in other configuration files or are defined internally by PolicyCenter.
Worksheet	A page that can be shown in the workspace, the bottom pane of the web interface. The main advantage of worksheets is that they can be viewed at the same time as regular pages. This makes them appropriate for certain kinds of detail pages such as creating a new note.
Popup	A page that appears on top of another page, and that returns a value to its invoking page. Popups allow users to perform an interim action without leaving the context of the original task. For example, a page that requires the user to specify a contact person could provide a popup to search for the contact. After the popup closes, PolicyCenter returns the contact to the invoking page.
Forward	A location with zero screens. Since it has no screens, it has no visual content. A Forward must immediately forward the user to some other location. Forwards are useful as placeholders and for indirect navigation. For example, you might want to link to the generic Desktop location. This would then forward the user directly to the specific Desktop page (for example, Desktop Activities) most appropriate for that kind of user.
Location group	A collection of locations. Typically a location group is used to provide the structure and navigation for a group of related pages. PolicyCenter can automatically display the appropriate menus and other interface elements that allow users to navigate among these pages.

Widgets

A *widget* is an element that PolicyCenter can render into HTML. PolicyCenter then displays the HTML visually. Buttons, menus, text boxes, and data fields are all examples of widgets. There are also a few widgets that you cannot see directly, but that otherwise affect the layout of widgets that you can see.

For most locations, a *screen* is the top-most widget. It represents a single HTML page of visual content within the main work area of the PolicyCenter interface. Thus, a screen typically contains other widgets. You can reuse a single screen in more than one location.

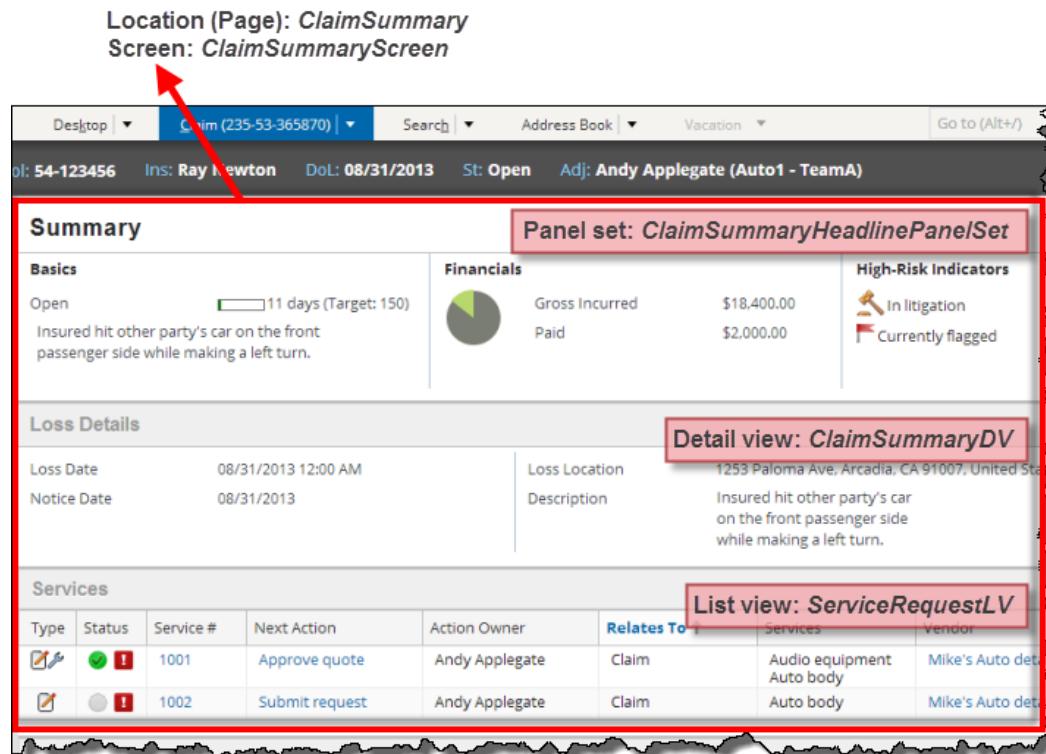
The following diagram shows a possible widget hierarchy:



Identifying PCF Elements in the User Interface

To modify a particular page in PolicyCenter, you must first understand how it is constructed. This includes understanding the PCF elements which compose the page, what files define the PCF elements, and how they are pulled together.

For example, consider the Claim Summary page within ClaimCenter. If you look at this page in the ClaimCenter interface, you cannot immediately tell how it is constructed. If you want to modify this page, some of the important things to know about it are illustrated in the following annotated diagram:



This diagram shows:

- The location is a page named `ClaimSummary`.
- The page contains a screen named `ClaimSummaryScreen`.
- The screen contains a “panel set” widget named `ClaimSummaryHeadlinePanelSet`.
- The screen contains a “detail view” widget named `ClaimSummaryDV`.
- The screen contains a “list view” widget named `ServiceRequestLV`.

PolicyCenter provides the following tools that allows you to view the structure of any page and to see which PCF elements it uses:

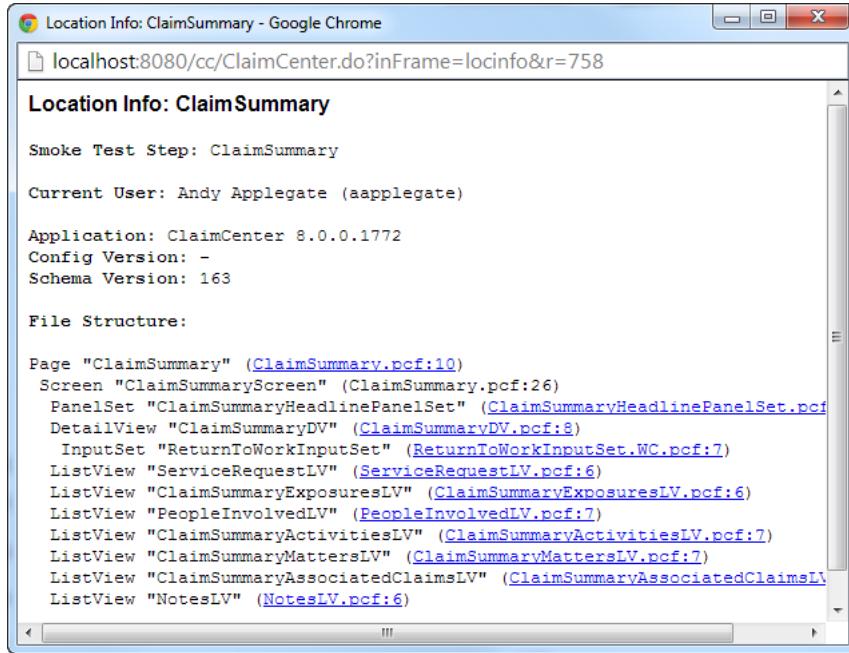
- Location Info
- Widget Inspector

To enable these tools, the `EnableInternalDebugTools` configuration parameter must be set to `true`.

Location Info

The **Location Info** window shows you information about the construction of the page you are viewing. It includes the location name, screen names, and high-level widgets defined in the page, and the names of the PCF files in which they are all defined. Typically, the widgets that appear in this window are the ones that are defined in separate files, such as screens, detail views, list views, and so on. The **Location Info** is most useful if you are making changes to a page as it tells you which files you need to modify.

To view the location information for a particular page, go to that page in the PolicyCenter interface, and then press ALT+SHIFT+I. This pops up the **Location Info** window for the active page. For example, the following is the **Location Info** window for the ClaimCenter Claim Summary page:



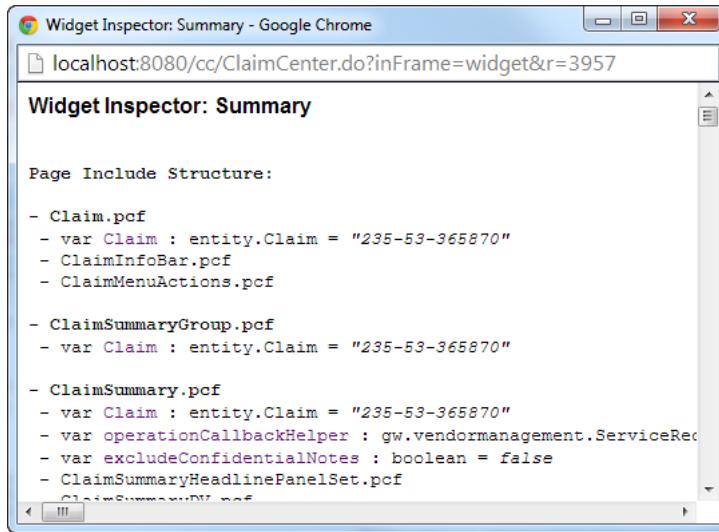
With this information, you can see:

- The location is a page named `ClaimSummary`, defined in the `ClaimSummary.pcf` file on line 10.
- The page contains a screen named `ClaimSummaryScreen`, defined in the `ClaimSummary.pcf` file on line 26.
- The screen contains one detail view widget, and multiple list view widgets, each defined in a different file.

Widget Inspector

The **Widget Inspector** shows detailed information about the widgets that appear on a page. This includes the widget name, ID, label text, and the file in which it is defined. The widget information is most useful during debugging a problem with a page. For example, suppose that a defined widget does not appear on a page. You could then look at the widget information to determine whether the widget exists (but perhaps is not visible) or does not exist at all.

To view the widget inspector for a particular page, go to that page in the PolicyCenter interface, and then press ALT+SHIFT+W. This pops up the **Widget Inspector** window for the active page. For example, the following graphic shows the **Widget Inspector** window for the ClaimCenter **Claim Summary** page:



The first part of the window shows the variables and other data objects defined in the page. After that, all of the widgets on the page are listed in hierarchical order.

Getting Started Configuring Pages

This section provides a brief introduction to the most useful and common tasks that you might need to perform during page configuration. It covers the following topics:

- Finding an Existing Element To Edit
- Creating a New Standalone PCF Element

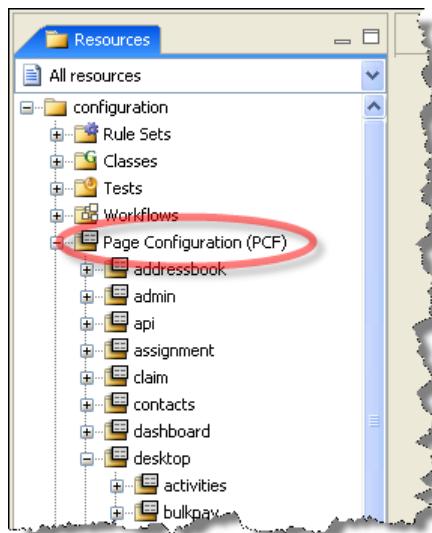
Finding an Existing Element To Edit

The first step in modifying the PolicyCenter interface is finding the PCF element that you want to edit, whether this is a page, a screen, or a specific widget. There are several ways to do this:

- Browse the PCF Hierarchy
- Find an Element By ID

Browse the PCF Hierarchy

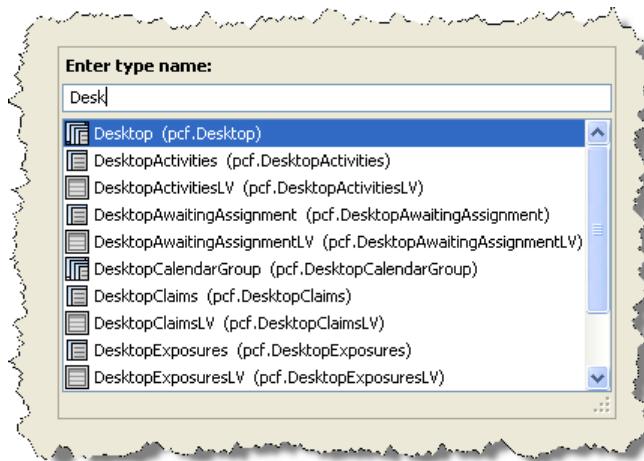
You can browse the PCF elements under the **Page Configuration** folder in Guidewire Studio:



These elements are arranged in a folder hierarchy that is related to how they appear in the PolicyCenter interface. For example, the **admin**, **claim**, and **dashboard** folders generally contain PCF elements that are related to the **Administration**, **Claim**, and **Dashboard** pages within ClaimCenter.

Find an Element By ID

If you know the ID of the element, such as by using the location info or widget inspector windows, you can find it within Studio. Press **CTRL+N** to open the **Find By Name** dialog box, and then start typing the ID of the element. As you type, PolicyCenter displays a list of possible elements that match the ID you are entering.



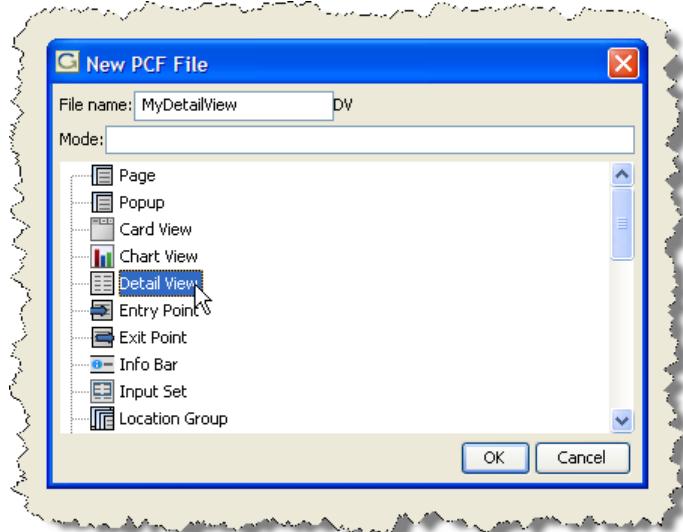
After you see the one you want, click on it and PolicyCenter opens the file in the PCF Editor.

Creating a New Standalone PCF Element

You can create a new PCF standalone element in Guidewire Studio. Each standalone element is stored in its own file.

To create a new standalone element

1. Browse the **Page Configuration** folder in the **Project** window, and locate the folder under which you want to create your new element.
2. Right-click on that folder, and then click **New → PCF File**. (You can also click **New → PCF Folder** to create a new folder). The **New PCF File** dialog appears.



3. In the **File name** text box, type the name of the element.
4. Click the type of element to create. If an element has a naming convention, it is shown next to the **File name** text box. For example, the name of a detail view must end with **DV**.
5. Click **OK**, and the new element is created and opened for editing in Studio.

Modifying Style and Theme Elements

Changing or Adding Images

Images used in the application reside in `PolicyCenter/modules/configuration/webresources/themes/Titanium/resources/images`. Images can be switched by replacing an existing image file with one of the same name. We recommend ensuring that replaced images are the same size as the original to avoid sizing issues.

To add a new image, place it in this folder or one of its children, then reference it as appropriate.

To update your application with these new images, run `gwpc update-theme` from the command line.

Overriding CSS

To override specific CSS classes, make edits to `PolicyCenter/modules/configuration/webresources/themes/Titanium/resources/theme_ext.css`. Changes in this file override other CSS properties in your application.

Changing Theme Colors

Guidewire applications are themed using SASS technology. To make significant changes to the style of your application, see “Advanced Re-Theming” on page 312. However, you can change the theme colors of your application by following these steps:

1. Open `PolicyCenter/ThemeApp/packages/titanium/sass/var/Component.scss` in a text editor. This is where all of the PolicyCenter colors are defined.
2. You can modify the definition to be any other hexadecimal color, or to be relative to another. For example, `$base-light-color` takes the `$base-color` and lightens it appropriately across the application.
3. After making changes, run `gwpc update-theme` from the command line. This incorporates your changes into the CSS generated by the SASS Theme.

For more information about SASS, visit <http://sass-lang.com>.

Advanced Re-Theming

PolicyCenter uses SASS to define a robust set of styling rules for ensuring a consistent look across the application. To make more detailed styling and theme changes, we recommend referencing the SASS documentation for details. There are, however, a few things specific to the Guidewire implementation:

- You cannot create a new theme and apply it to the application. All changes to the styling need to be made in the current theme definition, under `PolicyCenter/ThemeApp/packages/titanium/sass/`.
- SASS condenses its CSS definition for performance purposes. To see the expanded, debuggable CSS in your web development tool, run the command `gwpc dev-deploy-web-resources-debug` and refresh your browser.
 - `gwpc update-theme` re-condenses your CSS for production use.

Data Panels

This topic provides an introduction to the concepts and files involved in configuring the web pages of the PolicyCenter user interface.

This topic includes:

- “Panel Overview” on page 313
- “Detail View Panel” on page 313
- “List View Panel” on page 318

Panel Overview

A *panel* is a widget that contains the visual layout of the data to display in a screen. There are several types of panels:

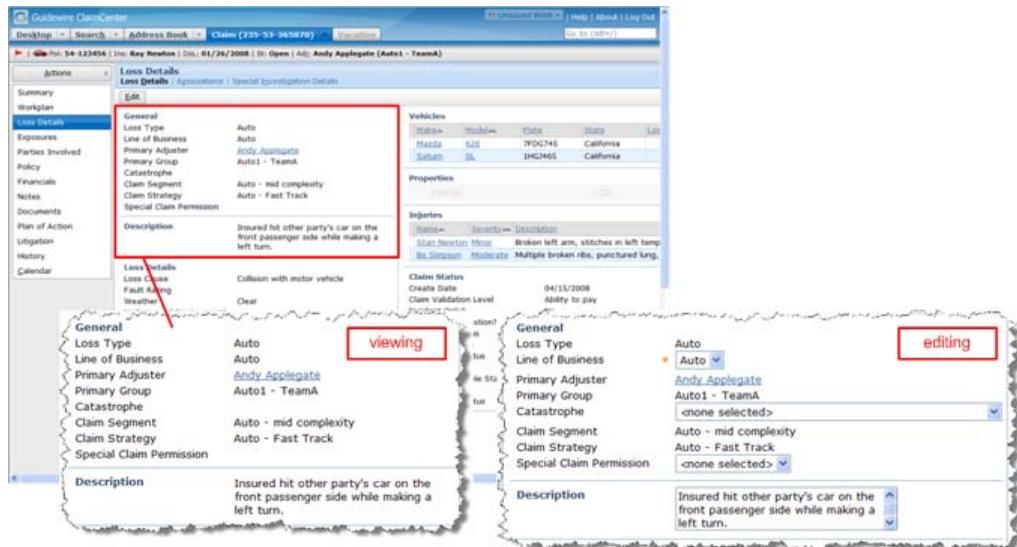
- Detail View Panel – A series of widgets laid out in one or more columns.
- List View Panel – A list of array objects, or any other data that can be laid out in tabular form.

You can place as many panels in a screen as you like, dividing the screen into one or more areas.

Detail View Panel

A *detail view* is a panel that is composed of a series of data fields laid out in one or more columns. It can contain information about a single data object, or it can include data from multiple related objects. Any input widget can appear within a detail view.

The following is an example of a detail view as it appears both as it is being viewed and as it is being edited:

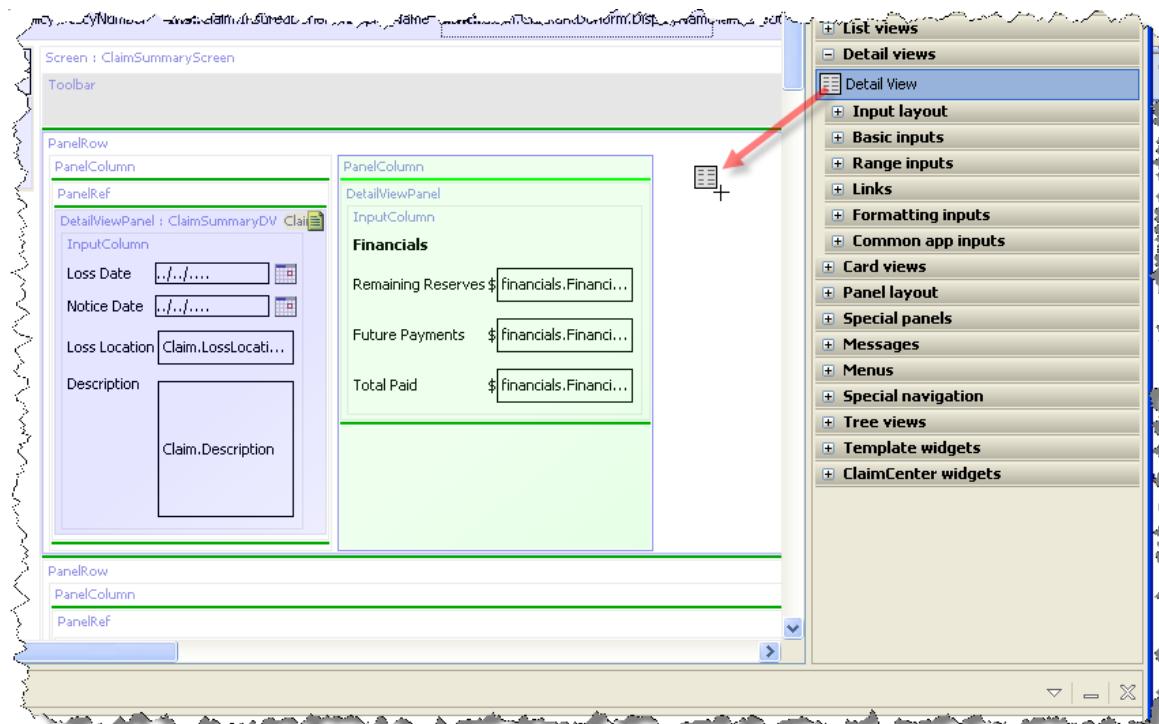


You can do the following:

- Define a Detail View
- Add Columns to a Detail View
- Format a Detail View

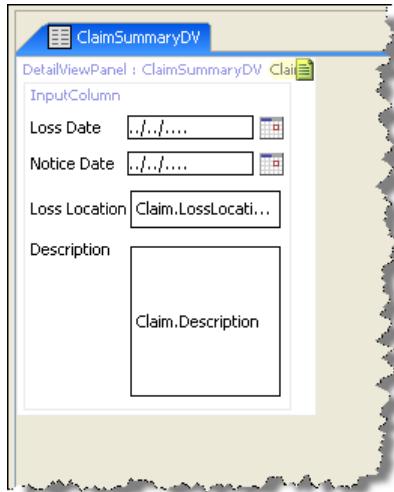
Define a Detail View

Define a detail view by dragging the Detail View element onto the PCF canvas. You can place the element anywhere a green line appears. For example:

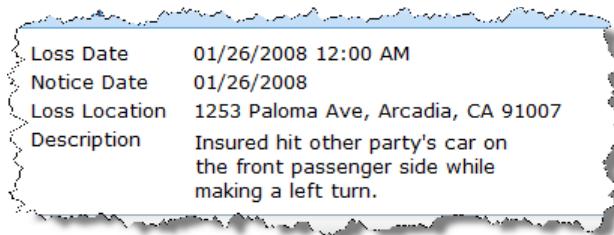


The `id` attribute is required; it identifies the panel so that it can be referenced by other PCF elements. The ID must be unique, and it must end with the text string DV.

A detail view must contain at least one vertical column, defined by the `Input Column` element. The column contains the input widgets to display, as in the following example:



This definition produces the following detail view:



Add Columns to a Detail View

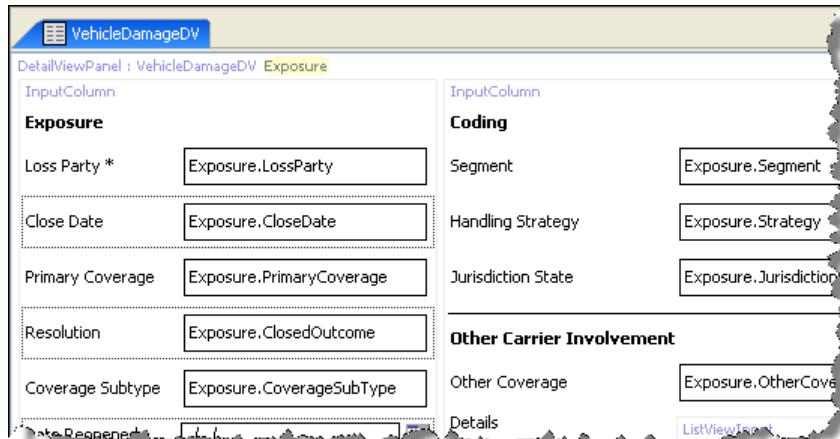
A detail view must contain at least one vertical column, but it can contain more. The following illustration shows detail views with one and two columns:

Column 1

Column 1

Column 2

A column is defined by the `Input Column` element. This element must appear at least once, to define the first column. To add additional columns, include the `Input Column` element multiple times. The following example defines a two-column detail view:



PolicyCenter automatically places a vertical divider between the columns.

The full definition of the previous example produces the following two-column detail view:

Exposure		Coding	
Loss Party	Insured's loss	Segment	Auto - low complexity
Primary Coverage	Liability - Property damage	Handling Strategy	Auto - Fast Track
Coverage Subtype	Liability - Property Damage - Vehicle	Jurisdiction State	California
Coverage			
Adjuster	Andy Applegate		
Group	Auto1 - TeamA		
Status	Open		
Create Date	04/15/2008		
Statistical Line	-		
Validation Level			
Claimant		Financials	
Claimant	Ray Newton	Remaining Reserves	-
Type	Owner of other vehicle	Future Payments	-
		Total Paid	-
		Total Recoveries	-
		Net Total Incurred	-

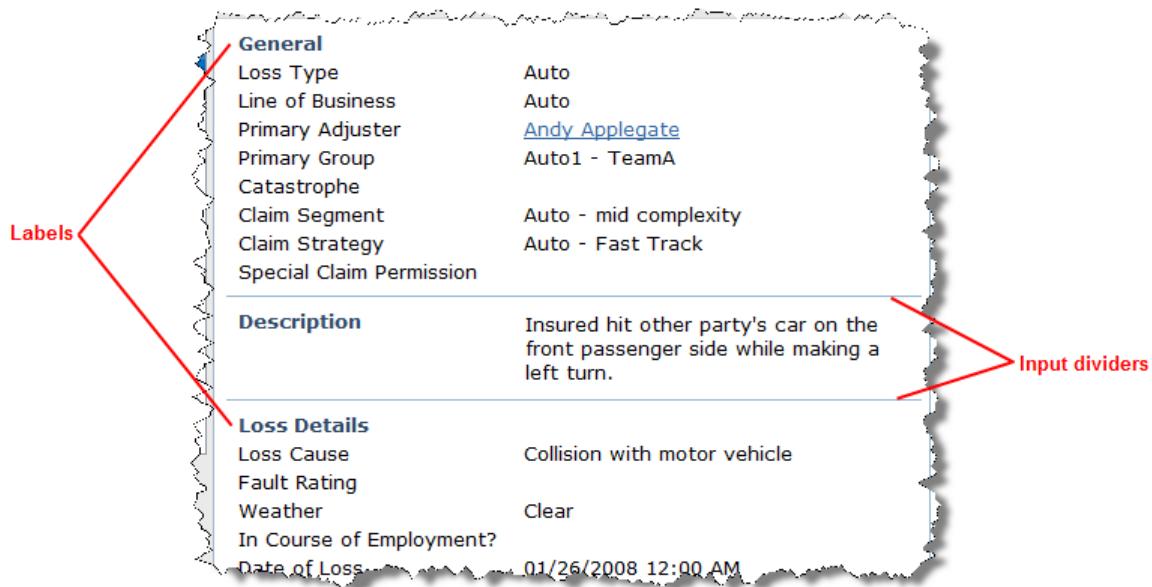
vertical divider

Format a Detail View

You can add the following formatting options to a detail view:

- Label
- Input Divider

These are illustrated in the following diagram:



Label

A label is bold text that acts as a heading for a section of a detail view. All input widgets that appear after a label are slightly indented to indicate their relationship to the label. The indenting continues until another label appears or the detail view ends. Thus, you cannot manually end a label indenting level at any point that you choose.

Include a label with the Label element:

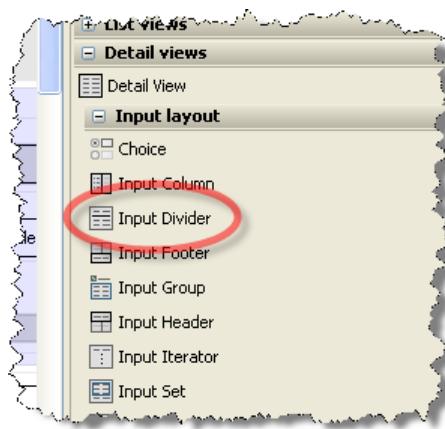


Set the `label` attribute to the display key to use for the label.

Input Divider

An input divider draws a horizontal line across a detail view column. You can place an input divider wherever you like between other elements.

Include an input divider with the `Input Divider` element:



List View Panel

A *list view* is a panel that displays rows of data in a two-dimensional table. The data can be an array of entities, results of a database query, reference table rows, or any other data that can be represented in tabular form.

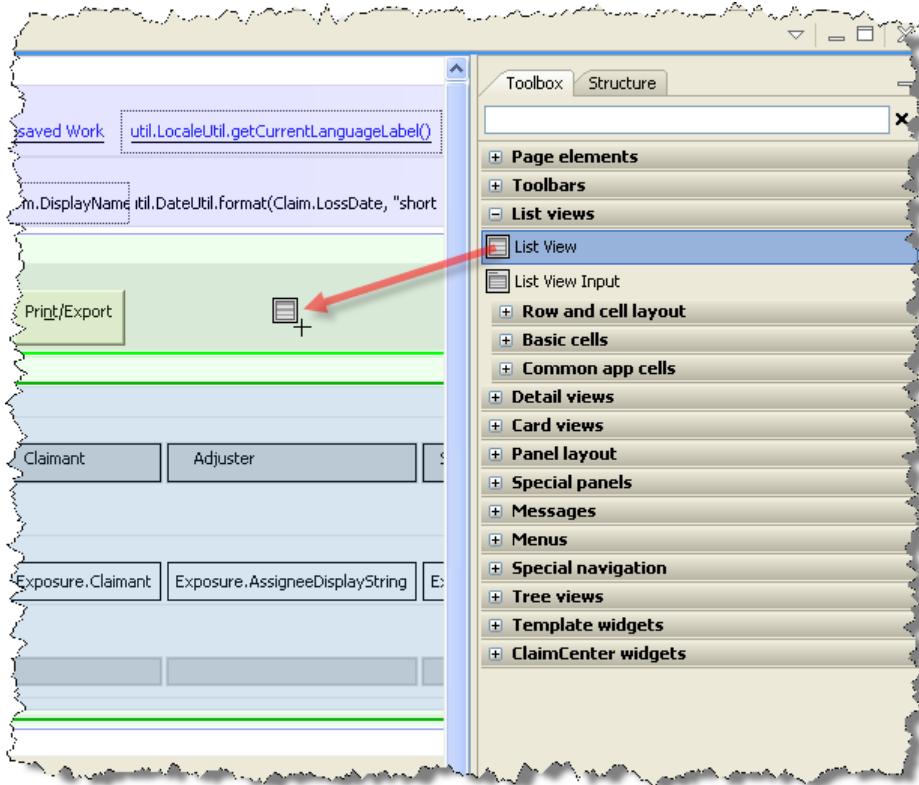
In most cases, data is viewed in list views and then edited in detail views. However, there are some places—for example, in the ClaimCenter financial transaction entry screens—in which it makes more sense to edit a list of items in place. For this purpose, you can make a list view editable so that you can add or remove rows, or modify cells of data.

The following is an example of a list view:

#	Type	Coverage	Claimant	Adjuster	Status	Remaining Reserves	Future Payments	Paid
<input type="checkbox"/>	1 Vehicle	Collision	Ray Newton	Andy Applegate	Open	\$400.00	-	\$500.00
<input type="checkbox"/>	2 Med Pay	Medical payments	Stan Newton	Andy Applegate	Open	\$2,000.00	-	\$1,500.00
<input type="checkbox"/>	3 Vehicle	Liability - Property damage	Bo Simpson	Andy Applegate	Open	\$5,000.00	-	-
<input type="checkbox"/>	4 Bodily Injury	Liability - Auto bodily injury	Bo Simpson	Carla Levitt	Open	\$9,000.00	-	-

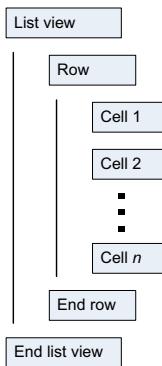
Define a List View

Define a list view by dragging the **List View** element onto the PCF canvas. You can place the element anywhere a green line appears. For example:

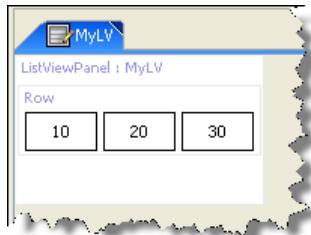


The `id` attribute is required; it identifies the panel so that it can be referenced by other PCF elements. The ID must be unique, and it must end with the text string `LV`.

A list view contains one or more *rows*, each containing one or more *cells*. The structure of the simplest one-row list view is illustrated below:



To define the rows and cells of the list view, use `Row` and `Cell` elements. Each occurrence of `Row` starts a new row, and each `Cell` creates a new column within the row. The following example creates a one-row, three-column list view:



The `id` attribute of a `Cell` element is required. It must be unique within the list view, but does not need to be unique across all of PolicyCenter. The `value` attribute contains the Gosu expression that appears within the cell. In the previous example, the value of each cell is set to 10, 20, and 30, respectively. You can set other attributes of a `Cell` to control formatting, sorting, and many other options.

This simple example demonstrates the basic structure of a list view. However, you will almost never use a list view with a fixed number of rows. The more useful list views iterate over a data set and dynamically create as many rows as necessary. This is illustrated in “Iterate a List View Over a Data Set” on page 321.

A list view requires a toolbar so that there is a place to put the paging controls, as well as any buttons or other controls that are necessary.

You can define a list view in the following ways:

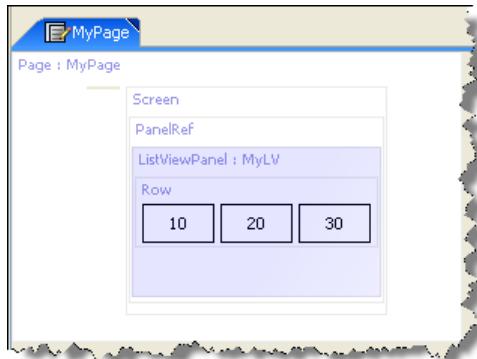
- Standalone
- Inline

Standalone

You can define a list view in a standalone file, and then include it in other screens where needed. This approach is the most flexible, as it allows you to define a list view once and then reuse it multiple times.

For example, suppose you define a standalone list view called `MyLV`.

You can then include this list view in a screen with the `PanelRef` element:

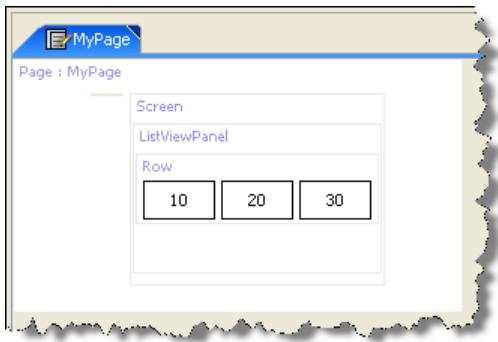


Set the `def` attribute of the `PanelRef` to the name of the list view; in this example, that is `MyLV`.

Inline

If a list view is simple and used only once, you can define it inline as part of a screen. This approach often makes it easier to create and understand a screen definition, as all of its component elements can be defined all in one place. However, an inline list view appears only where it is defined, and cannot be reused in other screens.

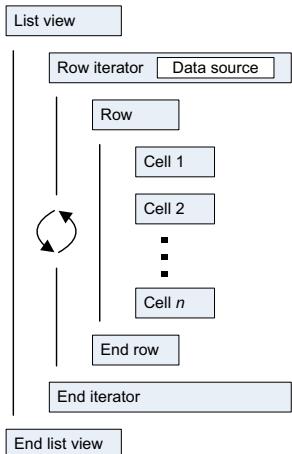
The following example defines an inline list view in a screen:



Iterate a List View Over a Data Set

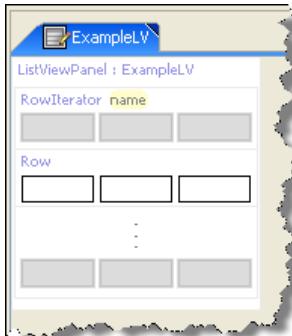
Most list views iterate over a data set and dynamically create a new row in the list for each record in the data set. The most common usage is showing an array of objects that belong to another object. For example, listing all activities that belong to a claim, or all users that belong to a group.

To construct a list view that iterates over a data set, use a *row iterator*. The structure of this kind of list view is illustrated in the following diagram:



The row iterator specifies the data source for the list. For each record in the data source, the iterator repeats the row (and other elements) defined within it.

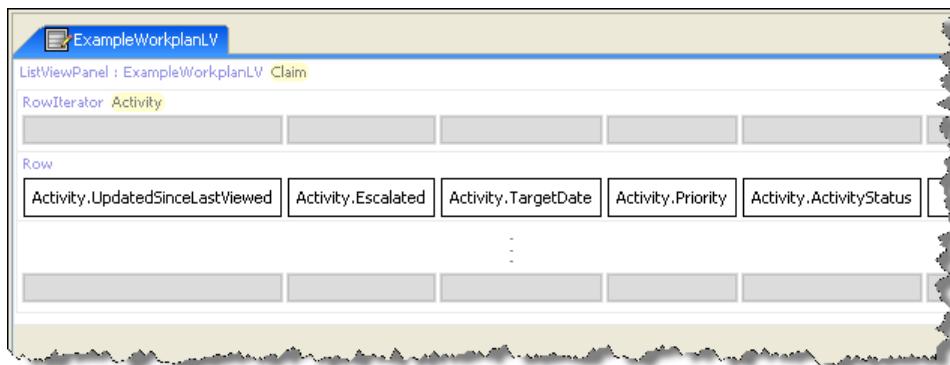
Define a row iterator with the **Row Iterator** PCF element. For example:



The **value** attribute of the **Row Iterator** specifies the data source, such as an array of entities or the results of a query. For more information on setting a data source, see “Choose the Data Source for a List View” on page 322.

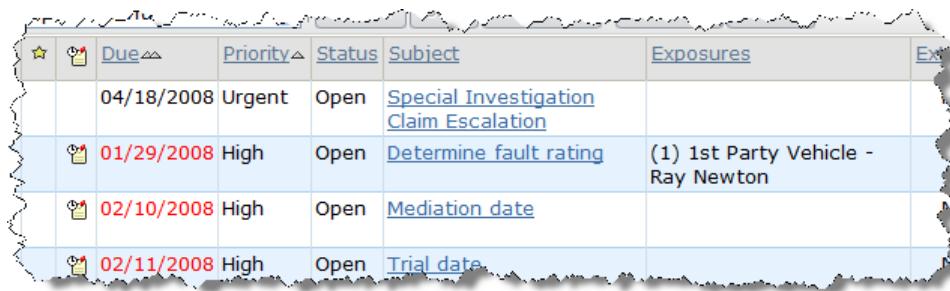
The `elementName` attribute is the variable name that represents the “current” row in the list. You can use this variable anywhere within the row iterator as the root object that refers to the current row.

Consider the following example, in which a `Claim` variable represents a claim:



To iterate over the array of activities in the claim, this list view creates a row iterator whose `value` attribute is `Claim.Activities`. For each activity in this array, the iterator creates a row with multiple cells. The `elementName` attribute of the iterator is `Activity`; it represents the current row, and is used to get the values of the `Activity` object’s fields in each cell.

This example produces the following list view:



	Due	Priority	Status	Subject	Exposures	Ex...
	04/18/2008	Urgent	Open	Special Investigation Claim Escalation		
	01/29/2008	High	Open	Determine fault rating	(1) 1st Party Vehicle - Ray Newton	
	02/10/2008	High	Open	Mediation date		
	02/11/2008	High	Open	Trial date		

Choose the Data Source for a List View

List views use different kinds of data sources to support different application requirements. The simplest data source is an array field on an entity type. An array field generally has a limited set of items that do not require a database query to retrieve. For example, the list of exposures for a claim is relatively short and is retrieved from the database as part of the overall claim, without a separate database query.

Other data sources for a list view involve a query and are more complex. This is especially true for search results or lists of items (activities, claims, and so on) on the Desktop. For example, a query as the source for a list view could be “all activities assigned to the current user that are due today or earlier.”

You specify the data source for a list view with the `value` property of the row iterator for the list view.

Source	Description
Array field	An <i>array field</i> on an entity type is identified in the <i>Data Dictionary</i> as an array key. For example, the <code>Officials</code> field on a <code>Claim</code> is an array key. Thus, you can define a list view based on <code>Claim.Officials</code> . In this case, each official listed on a specified claim is shown on a new row in the list view. You can also define your own custom Gosu methods that return array data for use in a list view. The method must return either a Gosu array or a Java list (<code>java.util.List</code>).
Query processor field	A <i>query processor field</i> on an entity type is identified in the <i>Data Dictionary</i> as a derived property returning <code>gw.api.database.IQueryBeanResult</code> . It represents an internally-defined query, and usually provides a more convenient and efficient way to retrieve data. For example, the <code>Claim.ViewableNotes</code> field performs a database query to retrieve only the notes on a claim that the current user has permission to view. This is more efficient than using the <code>Claim.Notes</code> array field, which loads both viewable and non-viewable notes and filtering the non-viewable ones out later.
Finder method	A <i>finder method</i> on an entity type is similar to a query processor field, except that it is not defined as field in the <i>Data Dictionary</i> . Instead, a finder method is an internally-defined Java class that performs an efficient query on instances of an entity type. For example, the <code>Activities</code> page of the <code>Desktop</code> uses a list view based on the finder method <code>Activity.finder.getActivityDesktopViewsAssignedToCurrentUser</code> .
Query builder result	A <i>query builder result</i> uses the result of an SQL query. For more information, see “Query Builder APIs” on page 127 in the <i>Gosu Reference Guide</i> .
Find expression query	A <i>find expression query</i> uses the result of an SQL query. Guidewire strongly recommends that you use query builder results as sources for list views instead of find expression queries.

List views behave differently depending on whether the source is an array or one of the query-backed sources.

Behavior	Array-backed list view	Query-backed list view
Loading data	The full set of data is loaded upon initially rendering the list view.	Only the data on the first page shown is fetched and loaded.
Paging	The full set of data is reloaded each time you move to a different page within the list view.	The query is re-run. Data is loaded only for the page that is viewable.
Sorting	The full set of data is reloaded each time the list view is sorted.	The query is re-run and sorted in the database. Therefore, you can sort only on columns that exist in the physical database, and not (for example) on virtual columns. Data is loaded only for the page that is viewable.
Filtering	The full set of data is reloaded each time the list view is filtered.	The query is re-run and filtered in the database. Therefore, you can filter only on columns that exist in the physical database, and not (for example) on virtual columns. Data is loaded only for the page that is viewable.
Editing	Paging, sorting, and filtering work as noted above, as long as any modified (but uncommitted) data is valid. Sorting and filtering can result in modified rows being sorted to a different page or filtered out of the visible list.	Paging, sorting, and filtering are disabled.
Best suited for	Short lists	Long lists
Additional notes	Do not use a query-backed editable list view in a wizard.	

Navigation

This topic provides an introduction to the concepts and files involved in configuring the web pages of the PolicyCenter user interface.

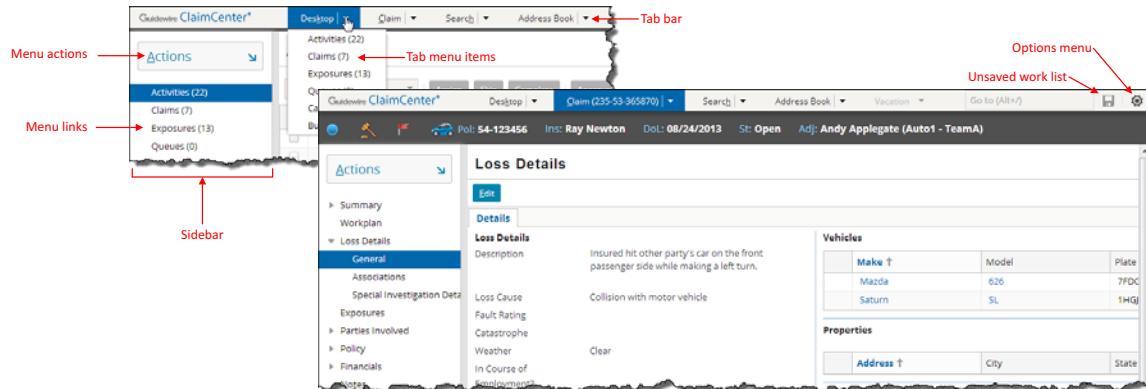
This topic includes:

- “Navigation Overview” on page 325
- “Tab Bars” on page 326
- “Tabs” on page 327

Navigation Overview

Navigation is the process of moving from one place in a Guidewire application interface to another. If you click on a link, you “navigate” to the location the link takes you.

A Guidewire application interface provides many elements that you use to navigate within the application. The following diagram identifies the most common navigation elements:

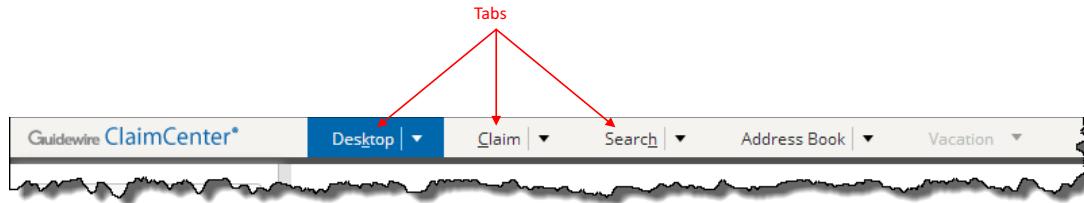


You can define the following types of navigation elements:

Tab bar	A set of tabs that run across the top of the application.
Tabs	Items in the tab bar that navigate to particular locations or show a drop-down menu.
Tab menu items	A set of links shown in the drop-down menu of a tab.
Menu links	Links in the sidebar that take you to other locations, typically within the context of the current tab.
Menu actions	Links under the Actions menu in the sidebar that perform actions that are typically related to what you can do on the current tab.

Tab Bars

A tab bar contains a set of tabs that run across the top of the application window, as in the following example:



You can do the following:

- Configure the Default Tab Bar
- Specify Which Tab Bar to Display
- Define a Tab Bar

You can also configure the individual tabs on a tab bar. For more information, see “Tabs” on page 327.

Configure the Default Tab Bar

PolicyCenter defines a default tab bar named `TabBar`. If no other tab bar is specified, then the default tab bar is used. However, if necessary, you can explicitly specify a different tab bar to show instead.

We recommend that you rely entirely on the default tab bar within the primary PolicyCenter application. You can customize the default tab bar to have it serve almost all of your needs. Consider defining a new tab bar only for special pages, such as entry points that have limited access to the rest of the application.

Specify Which Tab Bar to Display

You rarely need to explicitly specify a tab bar to display. Instead, you almost always rely on the default tab bar `TabBar`. However, to override the default and specify a different tab bar, set the `tabBar` attribute on the location group. For example, you could set it to `MyTabBar()`.

As you navigate to a location, PolicyCenter scans up the navigation hierarchy and checks whether a tab bar is explicitly set on a location group. If so, then that tab bar is used. If no tab bars are set, then the default tab bar is used.

For user interface clarity and consistency, we recommend that you set the tab bar only on the top-most location group in the hierarchy. However, a tab bar set on a child location group overrides the setting of its parent.

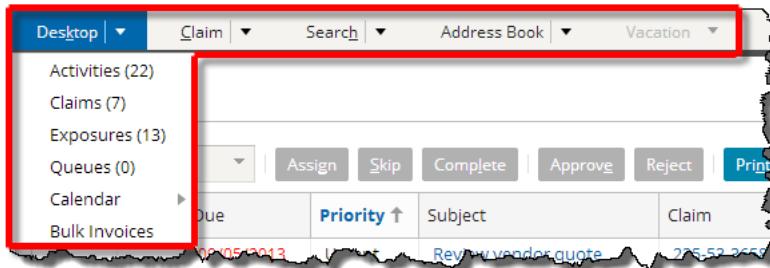
Define a Tab Bar

Define a tab bar with the `TabBar` PCF element. For example:



Tabs

Tabs are items in a tab bar that you can click on. A tab can be a single link that takes you directly to another location, it can be a drop-down menu, or it can be both. The following shows an example of tabs on a tab bar in ClaimCenter:



You can do the following:

- Define a Tab
- Define a Drop-down Menu on a Tab

Define a Tab

Define a tab by placing a `Tab` PCF element with a `Tab Bar`. For example:



The `action` attribute of a tab defines where clicking the tab takes you. For example, to go to the Desktop location, set the `action` attribute to `Desktop.go()`.

Define a Drop-down Menu on a Tab

A tab can contain a drop-down menu. As a tab has a menu, it shows the menu icon . Clicking this icon shows the menu items, while clicking the other parts of the tab performs the tab action.

Menu items on a tab are defined in the following ways:

- implicitly, using a location group
- explicitly, defined by `<MenuItem>` elements

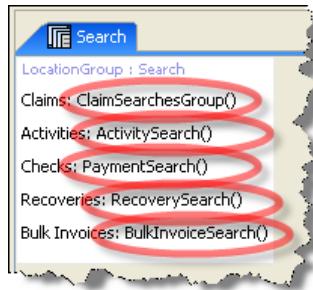
Define a Tab Menu From a Location Group

As the `action` attribute of a tab is a location group, PolicyCenter automatically creates menu items on the tab that correspond to the locations in that location group. For each location in the location group:

- a menu item is created in the tab
- the `label` attribute of the `Location Ref` is used as the label of the menu item
- the permissions of the location determine whether the menu item is available to the current user

For example, the action of the ClaimCenter **Search** tab goes to the **Search** location group. Its action attribute is defined as: `Search.Go()`.

This **Search** location group contains the `Location Ref` elements that appear as menu items on the tab:



This creates the menu items that appear on the **Search** tab:



Define a Tab Menu Explicitly

You can create a menu on a tab by explicitly defining `Menu Item` elements within the `Tab` definition. This method of creating a menu supersedes the automatic menu items derived from the location group. If you build a menu explicitly, PolicyCenter does not automatically add any other items to it.

Configuring Search Functionality

PolicyCenter provides a **Search** tab that you can use to search for specific entities. You can configure the **Search** tab to add new search criteria or modify or remove existing criteria. Configuring search functionality involves modifying the PolicyCenter data model through metadata definition files and modifying the PolicyCenter interface through page configuration files.

WARNING Guidewire strongly recommends that you consider all the implications before configuring the **Search** tab. Adding new search criteria can result in significant performance impacts, particularly in large databases. Guidewire recommends that you thoroughly test any search customizations for performance issues before you move them into a production database.

This topic includes:

- “Search Overview” on page 329
- “Database Search Configuration” on page 331
- “Free-text Search Configuration” on page 341

Search Overview

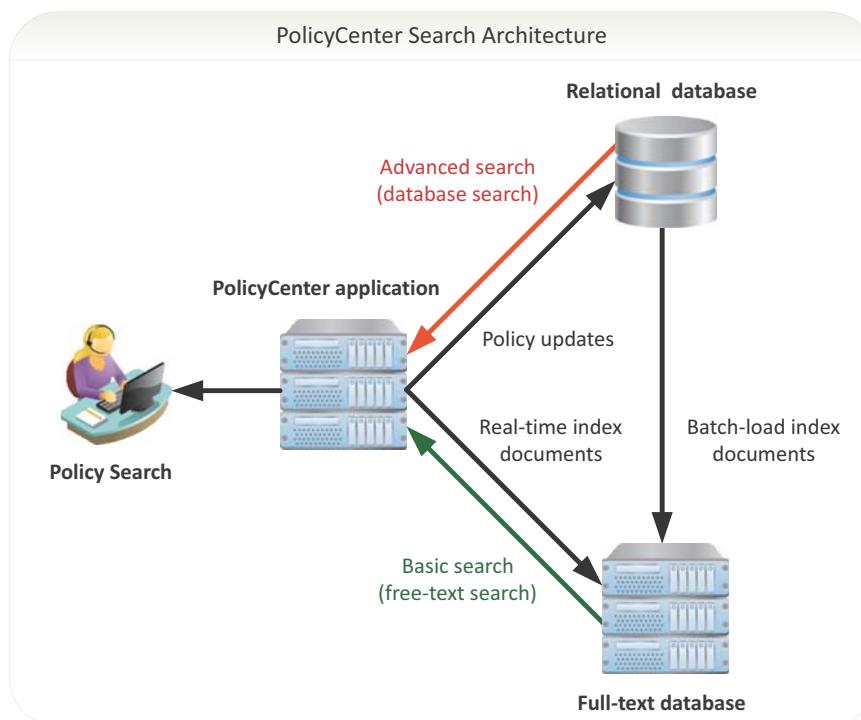
PolicyCenter provides two types of search:

- **Database search** – Searches the relational database for policies, accounts, producers, activities, and contacts by using Structured Query Language (SQL). You access these searches from the **Search** tab.
PolicyCenter also includes database search from screens besides those accessed through the **Search** tab. For example, you can do a database search for policy form patterns, policy locations, regions, and other entities and objects.
- **Free-text search** – Searches an external, full-text database for policies, by using the APIs of an external full-text search engine. PolicyCenter includes free-text search for policies and submissions. You access free-text search from the **Search Policies → Basic** screen.

Database search is fully enabled by default. Users can search policies, accounts, producers, activities, and contacts with database search. Users can choose to include archived policies with each database search request. The user interface for database search is known as *advanced search*.

Free-text search is available as an option that you must enable and configure. Users can search only for policies and submissions with free-text search. Free-text search results never include archived policies. The user interface for free-text search is known as *basic search*.

IMPORTANT Guidewire does not support configuring PolicyCenter for free-text search of entity types other than policies and submissions.



If you enable free-text search, the **Search Policies** screen displays two tabs:

- **Basic** – Displays fields on which to search policies and submissions. The **Basic** screen uses free-text search to return policies and submissions that match criteria users enter.
- **Advanced** – Displays fields on which to search policies and jobs. The **Advanced** screen uses database search to return policies and jobs that match criteria users enter.

If you enable free-text search, the **Search Policies** screen displays the **Basic** and **Advanced** search screens. Otherwise, the **Search Policies** screen displays the **Advanced** screen only.

Reasons users choose the **Basic** or **Advanced** screens include:

- Users want to search policies with commonly used criteria and receive results quickly, which the **Basic** screen provides.
- Users want to search policies with highly targeted criteria, which requires the **Advanced** screen.
- Users want to search policies with partial names, phonetic names, or sounds-like names, which the **Basic** screen provides.
- Users want to search archived policies, which requires the **Advanced** screen.
- Users want to search for specific job types, such as all policy changes bound within a certain period.
- An administrator hides the **Basic** search screen while the free-text batch load command runs, but users still can search policies with the **Advanced** screen.

See also

- “Database Search Configuration” on page 331
- “Free-text Search Configuration” on page 341

Database Search Configuration

This section describes how to configure database search. Topics include:

- “PolicyCenter Database Search Functionality” on page 331
- “Configuring PolicyCenter Database Search” on page 332
- “Working with Database Search Criteria in XML” on page 335
- “Working with Database Search Criteria in Gosu” on page 337

WARNING Guidewire strongly recommends that you consider all the implications before customizing the **Search** tab. Adding new search criteria can result in significant performance impacts, particularly in large databases. Guidewire recommends that you thoroughly test any search customizations for performance issues before you move them into a production database.

PolicyCenter Database Search Functionality

To search for a specific entity, select the **Search** tab from the PolicyCenter interface. In the base configuration, you can search for the following:

- Policies
- Accounts
- Producer Codes
- Activities
- Contacts

During a search, PolicyCenter uses only those fields on the form for which you enter data. For example, if you search for a **Policy** and enter a **Last Name** but not a **Policy Number**, PolicyCenter omits **Policy Number** from the search.

In the base configuration, PolicyCenter also includes database search from screens besides those accessed through the **Search** tab. For example, you can do a database search for policy form patterns, policy locations, regions, and other entities and objects.

For each search, PolicyCenter uses one of the following types of objects to encapsulate search criteria.

Object type	Description
Virtual entity	PolicyCenter uses a few search criteria objects that are <i>virtual entities</i> . A virtual entity has no underlying table in the PolicyCenter database. Rather, these are non-persistent entities that exist only within the session in which you use them. An example of a non-persistent entity as a search criteria object is <code>DocumentSearchCriteria</code> , defined in <code>search-config.xml</code> .
Gosu class	An example of a Gosu class as a search criteria object is the <code>AccountSearchCriteria</code> class, which Guidewire defines in <code>AccountSearchCriteria.gs</code> .

Every field on the **Search** screen maps to an attribute on the relevant search criteria entity. For example, in the **Activity** search screen:

- **Assigned To** maps to `AssignedUser` in the `ActivitySearchDV` PCF file.
- `AssignedUser` maps to `searchCriteria.SearchAssignedUser` in the same file.

- `SearchAssignedUser` maps to `gw.activity.ActivitySearchCriteria`, the Gosu class definition that contains the business logic to search on the assigned user field.

Configuring PolicyCenter Database Search

There are multiple locations in PolicyCenter code in which you can configure database search functionality. The file or files in which you configure search depends upon whether or not the search criteria object is defined as a virtual entity or a Gosu class.

Virtual Entities as Search Criteria

If the search criteria is a virtual entity, you can modify the criteria by modifying `search-config.xml` and a Gosu search criteria enhancement class. For example, `PolicySearchCriteria` is a virtual entity. To modify the search behavior, you modify `search-config.xml` or `PolicySearchCriteriaEnhancement.gsx`.

Gosu Classes as Search Criteria

If the search criteria is a Gosu class, you modify the search by modifying the Gosu search criteria class. For example, `AccountSearchCriteria` is a Gosu class. To modify the search behavior, you modify the `AccountSearchCriteria.gs` class.

Search for Entities and Objects

The following table lists the entity and object types for which you can search, and the specific file or files in the base configuration. The Location in User Interface column describes one location in which you can search for this object. You can access certain objects from more than one location.

Location in user interface	Search for entity or object	Configure search criteria object in...
Search Policies screen from the Search tab	<code>PolicyPeriodSummary</code>	<code>PolicySearchCriteria</code> virtual entity <code>PolicySearchCriteriaEnhancement.gsx</code>
Search Accounts screen from the Search tab	<code>AccountSummary</code>	<code>AccountSearchCriteria.gs</code>
Search Accounts screen from the Search tab	<code>AccountSummary</code>	<code>AccountSearchCriteria700.gs</code>
Search Activities from the Search tab	<code>Activity</code>	<code>ActivitySearchCriteria.gs</code>
Account File Claims screen from Claims menu link in the Sidebar of an account	<code>ClaimSet</code> with array to <code>Claim</code> objects in a particular date range.	<code>ClaimSearchCriteria.gs</code>
Account File History screen from History menu link in the Sidebar	<code>History</code>	<code>HistorySearchCriteria.gs</code>
Industry Code search popup from the Search Accounts screen	<code>IndustryCode</code>	<code>IndustryCodeSearchCriteria.gs</code>
Search Contacts screen from the Search tab	<code>Contact</code>	<code>ContactSearchCriteria</code> virtual entity <code>ContactSearchCriteriaEnhancement.gsx</code>
Groups screen from the Administration → Users & Security menu item	<code>Group</code>	<code>GroupSearchCriteria</code> virtual entity <code>GroupSearchCriteriaEnhancement.gsx</code>
Organizations screen from the Administration → Users & Security menu item	<code>Organization</code>	<code>OrganizationSearchCriteria</code> virtual entity <code>OrganizationSearchCriteriaEnhancement.gsx</code>
Notes screen from Notes menu link in the Sidebar of a policy	<code>Note</code>	<code>NoteSearchCriteria</code> virtual entity <code>NoteSearchCriteriaEnhancement.gsx</code>

Location in user interface	Search for entity or object	Configure search criteria object in...
Search Producer Codes screen from the Search tab	ProducerCode	ProducerCodeSearchCriteria virtual entity
		ProducerCodeSearchCriteria.gs
		ProducerCodeSearchCriteriaEnhancement.gsx
Users screen from the Administration → Users & Security menu item	User	UserSearchCriteria virtual entity
		UserSearchCriteriaEnhancement.gsx
Policy Form Patterns from the Administration → Business Settings menu item	FormPattern	FormPatternSearchCriteria.gs
Messages screen from the Administration → Monitoring menu item	Message	MessageSearchCriteria.gs
Tools → Documents screen of a policy	Document	DocumentSearchCriteria virtual entity
Building Class Code in the Details popup in the Buildings screen of a Businessowners job wizard	BOPClassCode	BOPClassCodeSearchCriteria.gs
Property Class Code in the Building popup of a Commercial Property job wizard	CPClassCode	CPClassCodeSearchCriteria.gs
Class Code in the Exposures screen of a General Liability job wizard	GLClassCode	GLClassCodeSearchCriteria.gs
Covered Employees → Class Code in the State Info screen of a Workers' Compensation job wizard	WCClassCode	WCClassCodeSearchCriteria.gs
Account File History screen of an account	History	HistorySearchCriteria.gs
Industry Code Search popup from the Search → Accounts menu item	IndustryCode	IndustryCodeSearchCriteria.gs
PolicyLocationSearchAPI web service	PolicyLocation	PolicyLocationBoundingBoxSearchCriteria.gs
The popup that appears when you click Search for Nearby Locations in the Reinsurance screen of a Businessowners policy file.	PolicyLocation	PolicyLocationSearchCriteria.gs
Risk Description search popup that appears when you add Spoilage to the Location Information → Additional Coverages screen in the Businessowners job wizard.	RiskClass	RiskClassSearchCriteria virtual entity
		RiskClassSearchCriteria.gs
Account File Related Accounts screen accessed from the Related Accounts link in the sidebar of an account	AccountSummary	SharedContactAccountSearchCriteria.gs
Location Information popup from the Location screen in a Workers' Compensation job wizard	TaxLocation	TaxLocationSearchCriteria virtual entity
		TaxLocationSearchCriteria.gs
Territory Code Search popup that appears when you set the Businessowners Line Territory Code in the Location Information screen of a Businessowners job wizard.	DBTerritory	TerritoryLookupCriteria virtual entity
		TerritoryLookupCriteria.gs
Search Accounts popup from the Alt Billing Account picker in the Payments screen of a job wizard	BCBillingAccountSearchResult Gosu class	BillingAccountSearchCriteria

Location in user interface	Search for entity or object	Configure search criteria object in...
For forms, coverage patterns, and policy holds	ClausePattern	ClausePatternSearchCriteria.gs
Additional Coverages → Add Coverages button on the Vehicle Information popup in a Business Auto policy job wizard.	Zone	PCZoneSearchCriteria.gs
Search for Regions popup that appears when you click Add Hold Region on the Hold Regions tab of a New Policy Hold.	UWIssueType	UWIssueTypeSearchCriteria.gs
Issue Type Search popup that appears when you click to add a Type in a new authority profile. Access authority profiles from Admin → Users & Security → Authority Profiles	RateBook	RateBookSearchCriteria.gs This is a feature of Guidewire Rating Management.
Rate Routines screen from Administration → Rating menu	CalcRoutineDefinition	RateRoutineSearchCriteria.gs This is a feature of Guidewire Rating Management.
Rate Table Definitions screen from Administration → Rating menu	RateTableDefinition	RateTableDefinitionSearchCriteria.gs This is a feature of Guidewire Rating Management.
Search Programs from Reinsurance menu	RIProgram	ProgramSearchCriteria.gs This is a feature of Guidewire Reinsurance Management.
Search Agreements from Reinsurance → Agreements menu	LocationRisk	RIlocationRiskProximitySearchCriteria.gs This is a feature of Guidewire Reinsurance Management.
Search Agreements with Arrangement set to Treaty. Accessed through Reinsurance → Agreements.	RIAgreement	AgreementSearchCriteria.gs This is a feature of Guidewire Reinsurance Management.
Search Agreements with Arrangement set to Facultative. Accessed through Reinsurance → Agreements.	RIAgreement	FacultativeSearchCriteria.gs This is a feature of Guidewire Reinsurance Management.

Search Criteria Abstract Classes

The `EntitySearchCriteria` abstract class provides a standard way of searching for entities, such as the `BOPClassCode` entity. In general, the search criteria classes for entities extend this class. For the `BOPClassCode` entity, the `BOPClassCodeSearchCriteria` class extends the `EntitySearchCriteria` class.

The `SearchCriteria` abstract class is a less restrictive superclass that provides a standard way of searching for objects. In general, search criteria for objects other than entities extend this class.

If the Gosu search criteria class does not exist, you can create it.

For information on configuring search for contacts, see “Searching for Contacts” on page 87 in the *Contact Management Guide*.

Working with Database Search Criteria in XML

You use the `search-config.xml` file to define a mapping between the key data entities and certain non-persistent entities used for search criteria. The entries in the file have the following basic structure.

```
<CriteriaDef entity="name" targetEntity="name">  
    <Criterion property="attributename" targetProperty="attributename" matchType="type"/>  
</CriteriaDef>
```

The following table describes the XML elements in the `search-config.xml` file.

Element name	Subelement	Description
SearchConfig	CriteriaDef	Root element in <code>search-config.xml</code> .
CriteriaDef	Criterion	Specifies the mapping from a search criteria entity to the target entity on which to search. WARNING Do not add new <code>CriteriaDef</code> elements to <code>search-config.xml</code> . Instead, modify only the contents of existing <code>CriteriaDef</code> elements.
	Criterion	Specifies how PolicyCenter matches a column (field) on the search criteria to the query against the target entity. Use this element to perform simple matching only. Simple matches are criteria that match values in a single column of the same type in the target entity.

See also

- “The `<CriteriaDef>` Element” on page 335.
- “The `<Criterion>` Subelement” on page 336.

The `<CriteriaDef>` Element

A `<CriteriaDef>` element specifies the mapping from a search criteria entity to the target entity on which to search. For example, a `<CriteriaDef>` element can specify a mapping between a `DocumentSearchCriteria` entity and a `Document` entity. A `<CriteriaDef>` element uses the following syntax.

```
<CriteriaDef entity="entityName" targetEntity="targetEntityName">
```

These attributes have the following definitions.

<code><CriteriaDef></code> attribute	Required	Description
entity	Yes	Type name of the criteria entity
targetEntity	Yes	Type name of the target entity.

It is also possible to map a single search criteria entity to more than one target entity. For example, the `ClaimSearchCriteria` object has a `<CriteriaDef>` element associated with all of the following entities:

- `Policy`
- `PolicyPeriod`
- `Submission`
- `Cancellation`

Do not add new `<CriteriaDef>` elements into `search-config.xml`. Only modify the contents of existing ones. Also, do not remove a required base `CriteriaDef` element as this can introduce problems into your PolicyCenter installation.

WARNING Guidewire strongly recommends you do not remove `<CriteriaDef>` elements that exist in the base configuration.

A `<CriteriaDef>` element can have the following subelements.

<code><CriteriaDef></code> subelement	Description
<code>Criterion</code>	Performs simple, one-to-one mapping between a criteria entity attribute and a target entity attribute.

The `<Criterion>` Subelement

Within a `<CriteriaDef>` element you can define zero or more `<Criterion>` subelements. A `<Criterion>` element performs simple, one-to-one mapping between a criteria entity attribute and a target entity attribute. A `<Criterion>` element uses the following syntax.

```
<Criterion property="attributename"
           targetProperty="attributename"
           forceEqMatchType="booleanproperty"
           matchType="type"/>
```

These attributes have the following definitions.

<code><Criterion></code> attribute	Required	Description
<code>property</code>	•	The name attribute on the criteria entity. PolicyCenter uses this value to get the user's search term from the criteria entity.
<code>matchType</code>	•	This attribute is dependent on the data type of the <code>targetProperty</code> . See the following table for possible values.
<code>forceEqMatchType</code>		<p>The name of a Boolean property on the criteria entity:</p> <ul style="list-style-type: none"> If this attribute evaluates to <code>true</code>, then the Criterion uses an <code>eq</code> (equality) match. If this attribute evaluates to <code>false</code>, then the Criterion uses the <code>matchType</code> that the Criterion specifies to perform the match. <p>For example:</p> <pre><Criterion property="StringProperty" forceEqMatchType="FlagProperty" matchType="startsWith"/></pre> <p>This code uses a <code>startsWith</code> match for <code>StringProperty</code> unless the <code>FlagProperty</code> on the criteria entity is <code>true</code>, in which case, the match uses an <code>eq</code> match type.</p>
<code>targetProperty</code>		<p>The name attribute on the entity on which to search.</p> <p>IMPORTANT Do not use a virtual property on the entity as the search field.</p>

The following list describes the valid `matchType` values. For `String` objects, `matchType` case-sensitivity depends on the database, except for `startsWith` and `contains`, which are always case-insensitive.

Match type	Evaluates to	Use with data type	Comments
<code>contains</code>		<code>String</code>	IMPORTANT Guidewire strongly recommends that you avoid using the <code>contains</code> match type, if at all possible. The <code>contains</code> match type is the most expensive type in terms of performance.
<code>eq</code>	<code>equals</code>	<code>Numeric</code> or <code>Date</code>	

Match type	Evaluates to	Use with data type	Comments
ge	greater than or equal	Numeric or Date	
gt	greater than	Numeric or Date	
le	less than or equal	Numeric or Date	
lt	less than	Numeric or Date	
startsWith		String	The startsWith match type is very expensive in terms of performance, second only to the contains match type. Use startsWith with caution.

Performance Tuning for Specific Search Criteria

It is possible that adding an index can improve performance. The exact index to add depends on the database that you use and the details of the situation. Whenever you change the search criteria by adding or modifying a <Criterion> subelement, be certain that appropriate indexes are in place. Guidewire recommends that you consult a database expert.

For example, suppose that you add a column that is the most restrictive equality condition in your search implementation. In this case, consider adding an index with this column as the leading key column.

IMPORTANT For performance reasons, Guidewire strongly recommends that you avoid the contains match type if at all possible. The contains match type is the most expensive type in terms of performance.

Do Not Attempt to Modify the Required Search Properties

Guidewire divides the main search screens into required and optional sections. Guidewire has carefully chosen the properties in the required section to enhance performance. Therefore, do not change which properties are required properties. Adding your own required search criteria can cause performance issues severe enough to bring down a production database.

In addition, Guidewire has carefully chosen the match types of the existing required properties, due to restrictions on configuring fields on tables that are joined to the search table. Therefore, do not change the match types of existing required fields.

WARNING For performance reasons, Guidewire expressly prohibits the addition of new required fields or changing the match type of existing required fields in the PolicyCenter search screens.

Working with Database Search Criteria in Gosu

In the base PolicyCenter configuration, Guidewire provides Gosu classes to configure database search for a number of entity types. The following table lists some entity types for which users can search and the Gosu classes that you modify to configure the user interface for that type of search.

Entity type to search	Gosu search criteria class
Account	gw.account.AccountSearchCriteria
Activity	gw.account.ActivitySearchCriteria
Claim	gw.losshistory.ClaimSearchCriteria
IndustryCode	gw.product.IndustryCodeSearchCriteria
History	gw.history.HistorySearchCriteria

Guidewire also provides Gosu enhancements that you can use to configure searches for Policy and Contact objects.

```
gw.search.PolicySearchCriteriaEnhancement.gsx  
gw.plugin.Contact.ContactSearchCriteriaEnhancement.gsx
```

For more information on PolicyCenter support for Contact searches, see “PolicyCenter Support for Contact Searches” on page 110 in the *Contact Management Guide*.

For a complete list of entity types for which you can configure search criteria in Gosu, see “Configuring PolicyCenter Database Search” on page 332.

Note: To improve contact search, Guidewire adds the following fields to the Contact entity as denormalization fields from the Address entity:

- CityDenorm
- Country
- PostalCodeDenorm
- State

The following topics provide examples of how to modify the base configuration search:

- Example: Adding a New Optional Search Field to Activity Search
- Example: Modifying the PolicySearchCriteriaEnhancement Class

WARNING Guidewire strongly recommends that you consider all the implications before customizing any search configuration object. Adding new search criteria can result in significant performance impacts, particularly in large databases. Guidewire recommends that you thoroughly test any search customizations for performance issues before you move them into a production database.

Example: Adding a New Optional Search Field to Activity Search

To expand the search capabilities a Gosu search criteria object with additional optional search criteria, you need to do the following:

1. Declare the variable in the Gosu class definition file.
2. Add a field in the PCF that maps to that variable.
3. Incorporate the variable into the query defined within the Gosu search object class definition file.

The following example adds a new field to the Activity Search PCF, and then incorporates it into the query in `ActivitySearchCriteria`.

To add an optional search field to Activity Search

1. Open `ActivitySearchCriteria.gs` for editing in PolicyCenter Studio.

- a. Notice that PolicyCenter defines the following search criteria in the base configuration.

```
var _policyNumber : String as PolicyNumber  
var _accountNumber : String as AccountNumber  
var _overdueNow : Boolean as OverdueNow  
var _activityStatus : ActivityStatus as SearchedActivityStatus  
var _priority : Priority as SearchedPriority  
var _assignedUser : User as SearchedAssignedUser
```

- b. Add `_recurring : Boolean as Recurring` at the end of this list.

2. Add a display key for the Recurring field label:

- a. Navigate to **configuration** → **config** → **Localizations** → **lang** in Studio, and open the `display.properties` file.
 - b. Find the display key entries that begin with `Web.ActivitySearch`, and add the following line.

```
Web.ActivitySearch.Recurring = Recurring
```

3. Open ActivitySearchDV for editing in Studio. Add an Input widget in the optional section directly under the Overdue Now field. Enter the following values for this widget in the Properties area at the bottom of the screen.

editable	true
id	Recurring
label	displaykey.Web.ActivitySearch.Recurring
required	false
value	searchCriteria.Recurring

PolicyCenter defines variable searchCriteria for this PCF file under Required Variables. To see the definition of this variable, select the entire DetailViewPanel and then select the Required Variables tab. You see the following (in the base configuration).

name	searchCriteria
type	gw.activity.ActivitySearchCriteria

4. Open ActivitySearchCriteria.gs for editing. Add the following to the list of if statements in the makeQuery function.

```
if (Recurring != null) {  
    query.compare("Recurring", Equals , Recurring)  
}
```

5. Stop and restart the application server.

6. Test your search screen by doing the following:

- a. Open PolicyCenter and navigate to the Search → Search Activities screen.

- b. Perform an activity search, setting Recurring to Yes.

If by chance, your installation does not include a recurring activity, you can create one:

- a. Open a currently active policy.

- b. Select New Activity from the Actions menu, set Recurring to Yes, and proceed from there.

Example: Modifying the PolicySearchCriteriaEnhancement Class

You can add additional search criteria to policy searches using PolicySearchCriteriaEnhancement.

1. Extend the PolicySearchCriteria object and add your search field to it as an extension column. The example adds a vehicle VIN column to the PolicySearchCriteria extension (PolicySearchCriteria.etx).
2. Then, add the additional search criteria fields in PolicySearchScreen.pcf. The example modifies this PCF file and adds an optional Input widget for the VIN number. PolicyCenter displays this VIN field only if you perform a Policy search on a SearchObjectType of Policy (and not Cancellation, for example).
3. Finally, modify the PolicySearchCriteriaEnhancement.search method by adding the desired search criteria.

The following table lists the PolicyCenter files involved in modifying a base configuration Policy search.

Location	Contains...
search-config.xml	<p>CriterionDef elements for the following:</p> <ul style="list-style-type: none"> • Policy • PolicyPeriod • Submission • Cancellation • Reinstatement • PolicyChange • Renewal • Rewrite • RewriteNewAccount • Audit <p>PolicyCenter uses these for SearchObjectType in PolicySearchScreen.pcf.</p>
PolicySearchScreen.pcf	<p>SearchPanel: Properties</p> <pre>searchCriteria: new PolicySearchCriteria() search: searchCriteria.search()</pre> <p>SearchFor: Properties</p> <pre>value: searchCriteria.SearchObjectType</pre> <p>PolicySearchScreen also contains fields for all searchable criteria in the PolicyCenter Search Policies screen. If you modify the base configuration Policy search, then you need to add your additional search fields to this PCF file.</p>
PolicySearchCriteria.eti	<p>Base configuration data model definitions for standard fields on the PolicySearchCriteria object. These include the following:</p> <ul style="list-style-type: none"> • AccountNumber • JobNumber • State • <p>You cannot modify this base configuration file.</p>
PolicySearchCriteria.etc	<p>Base configuration enhancement fields on the PolicySearchCriteria object. These consist of the following:</p> <ul style="list-style-type: none"> • AssignedRisk • UWCompany <p>You need to modify this file and add your own search configuration criteria if you modify the Policy search configuration. This example adds a VIN column.</p>
PolicySearchCriteriaEnhancements.gsx	<pre>function search()</pre> <p>You need to modify the search method if you modify the Policy search configuration.</p>

To modify the policy search configuration

The following example adds a vehicle VIN field to the PolicyCenter Search Policies screen.

1. Open PolicySearchCriteria.etc for editing in Studio and add a column defined as follows.

Name	Value
name	VIN
type	vin
desc	VIN (vehicle identification number) of the vehicle.

2. Open PolicySearchCriteriaEnhancement.gsx in Studio for editing and add the following if statement to the existing search method.

```
function search() : IQueryBeanResult<PolicyPeriodSummary> {
    ...
    // If no additional criteria are required, this method will construct the query and execute
    // the select as above.
```

```

if (!meetsMinimumSearchCriteria())
    throw new DisplayableException(displaykey.Web.Policy.MinimumSearchCriteria);
// add this if statement to the search method
//
if (this.VIN != null) {
    // construct the query using the base product criteria fields
    var query = this.SummaryQuery
    var MAX = 10 //Number of results to return

    var subQuery = new Query<PolicyPeriodSummary>(PolicyPeriodSummary)
    var vehTable = subQuery.subselect("ID", CompareIn, PersonalVehicle, "BranchValue")
    vehTable.compare("Vin", Equals, this.VIN)
    query = query.intersect(subQuery)

    // Then execute the query and check whether it exceeds a max number of rows
    var result = query.select()

    if(result.getCountLimitedBy(MAX + 1) > MAX) throw "Too many results"
    return result
}
// end of added if statement
//
var result = this.performSearch()
return result
}

```

3. Add a display key for the VIN field label:

a. Navigate to **configuration** → **config** → **Localizations** → **lang** in Studio, and open the **display.properties** file.

b. Find the display key entries that begin with **Web.PolicySearch**, and add the following line.

Web.PolicySearch.VIN = VIN

4. Open **PolicySearchScreen.pcf** for editing in Studio.

5. Double-click to open **DatabasePolicySearchPanelSet.pcf**. On the **Advanced** tab, add an **Input** widget directly under the **As of Date** field. Enter the following values for this widget in the **Properties** area at the bottom of the screen.

editable	true
id	VIN
label	displaykey.Web.PolicySearch.VIN
required	false
value	searchCriteria.VIN
visible	searchCriteria.SearchObjectType == "Policy"

The **visible** attribute determines whether PolicyCenter displays the VIN field in the interface. Notice that it is dependent on the value set for **SearchObjectType**. You can optionally make the Gosu expression more complicated. For example, you can add a condition that PolicyCenter only display the VIN field if the Product code is Personal Auto.

6. Stop and restart PolicyCenter Studio.

7. Stop and restart the PolicyCenter application server.

8. Test your work by navigating to the **Search Policies** screen in PolicyCenter and enter various search criteria.

Free-text Search Configuration

This topic describes how to enable and configure free-text search for PolicyCenter. It includes:

- “Overview of Free-text Search” on page 342

- “Free-text Search System Architecture” on page 342
- “Enabling Free-text Search in PolicyCenter” on page 346
- “Configuring the Solr Extension for Integration with PolicyCenter” on page 347
- “Configuring the Free-text Batch Load Command” on page 350
- “Configuring Free-text Search for Indexing and Searching” on page 350
- “Configuring the Basic Search Screen for Free-text Search” on page 351
- “Modifying Free-text Search for Additional Fields” on page 352

See also

- “Free-text Search Setup” on page 89 in the *Installation Guide*
- “Free-text Search Integration” on page 593 in the *Integration Guide*

Overview of Free-text Search

Free-text search depends on a full-text search engine, the Guidewire Solr Extension. You can configure free-text search and the Guidewire Solr Extension for different kinds of operation:

- **External** – Supported in production and development environments, the Guidewire Solr Extension runs as a separate application in a different instance of the application server than the instance that runs PolicyCenter.
- **Embedded** – Supported only in development environments, the Guidewire Solr Extension runs automatically as part of PolicyCenter in the application server instance that runs PolicyCenter. With embedded operation, the Guidewire Solr Extension does not run as a separate application.

Free-text Search System Architecture

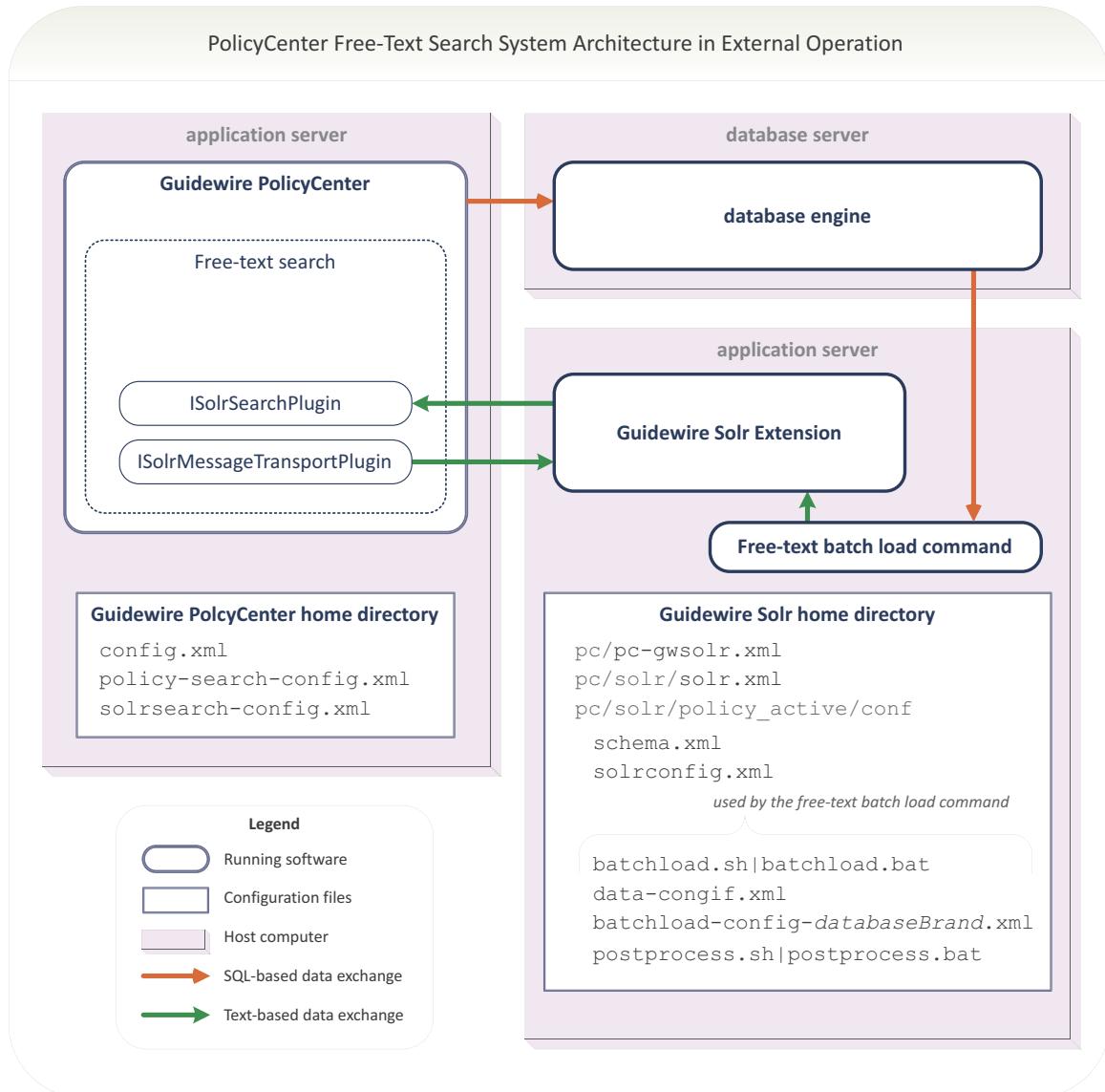
The system architecture of the Guidewire free-text search feature comprises the following components:

- The Guidewire PolicyCenter application
- The Guidewire Solr Extension, a modified version of the Apache Solr full-text search engine
- The Guidewire free-text batch load command
- The **Free-text Search** page on the **Server Tools** tab in the PolicyCenter application as an alternative during development to running the free-text batch load command

The components of the free-text search feature depend on configuration parameters and configuration files in two primary locations: the PolicyCenter home directory and a separate Guidewire Solr home directory.

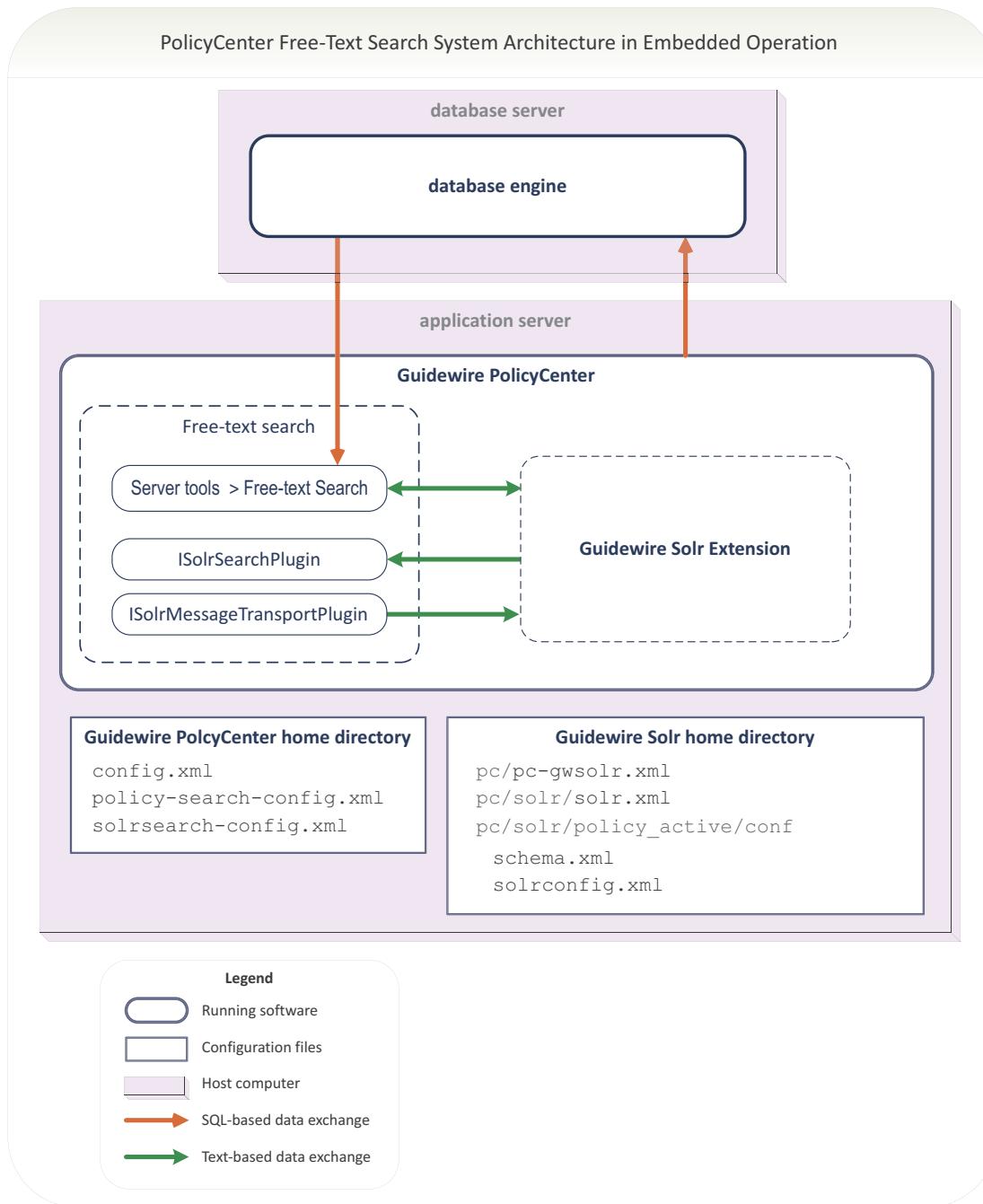
Free-text Search System Architecture in a Production Environment

The following diagram illustrates the system architecture for free-text search if you run PolicyCenter in a production environment. In a production environment, you must configure free-text search for external operation.



Free-text Search System Architecture in a Development Environment

The following diagram illustrates the system architecture for free-text search if you run PolicyCenter in a development environment. In a development environment, you can configure free-text search of embedded operation.



With embedded operation, the **Free-text Search** page on the **Server Tools** tab is available as an alternative to the free-text batch load command.

Free-text Search Configuration Parameters and Files

The free-text feature of PolicyCenter depends on configuration parameters and files in various locations.

Configuration Parameters and Files for Free-text Search in PolicyCenter

The following parameters and files help enable and configure free-text search in PolicyCenter:

- `FreeTextSearchEnabled` – Defined as a *configuration parameter* in `config.xml`, enables certain back-end components of free-text search to fully operate. The default value is `false`.
- `EnableDisplayBasicSearchTab` – Defined as a *script parameter*, enables the **Search Policies** → **Basic** screen for free-text search. You define script parameters initially through Studio but administer them on the **Script Parameters** page of the **Administration** tab in the application. The script `EnableDisplayBasicSearchTab` parameter has no effect if the `FreeTextSearchEnabled` configuration parameter is set to `false`.

Note: Set this script parameter to `false` before running the free-text search batch load command, and set it back to `true` after the batch command finishes. Setting the parameter to `false` prevents users from performing free-text searches while the batch process runs.

- `policy-search-config.xml` – Provides detailed configuration of the fields that free-text search extracts from the PolicyCenter database and sends to the full-text search database for indexing and searching.
- `solrserver-config.xml` – Configures how PolicyCenter works with the Guidewire Solr Extension, including connection information, and whether the mode of operation is external or embedded.

See also

- For details about the configuration parameter, see “Search Parameters” on page 75.
- For details about the script parameter, see “Script Parameters” on page 100.
- For details about the `solrserver-config.xml` configuration file, see “Configuring the Solr Extension for Integration with PolicyCenter” on page 347.

Configuration Files for the Guidewire Solr Extension

The following files configure the Guidewire Solr Extension, the full-text search engine that the free-text search feature depends upon. These configuration files control how the Guidewire Solr Extension loads data that PolicyCenter sends for indexing and how the Guidewire Solr Extension responds to search requests from PolicyCenter.

- `pc-gwsolr.xml` – Defines the Guidewire Solr home directory for the Guidewire Solr Extension.
- `solr.xml` – Defines the location in the Guidewire Solr home directory of the core for each searchable entity type in the Guidewire Solr Extension.
- `schema.xml` – Defines fields of data as known in the Guidewire Solr Extension.

See also

- “Configuring the Solr Extension for Integration with PolicyCenter” on page 347
- “Configuring Free-text Search for Indexing and Searching” on page 350

Configuration Files for the Free-text Batch Load Command

The following files configure the free-text batch load command. The command extracts data directly from the PolicyCenter database through native SQL commands and loads the extracted data into the Guidewire Solr Extension.

- `data-config.xml` – Specifies the location of the index documents that the free-text batch load command creates for the Guidewire Solr Extension to load. The file also specifies the mapping between fields in the index documents and fields defined in `search.xml`.
- `batchload-config-databaseBrand.xml` – Specifies working resources for the free-text batch load command. The file also contains the native SQL that the free-text batch load command uses to extract data from the database server.
- `batchload.sh/batchload.bat` – Runs the free-text batch load command. The file sets two environment variables:

- GWSOLR_HOME – The root of the Guidewire Solr home directory
- TARGET – The batchload-config-*databaseBrand*.xml file to use.
- postprocess.sh/postprocess.bat – Collates and compiles index documents for the Guidewire Solr Extension using data selected from the relational database.

See also

- “Configuring the Free-text Batch Load Command” on page 350

Enabling Free-text Search in PolicyCenter

Full-text search is disabled in the base configuration of PolicyCenter. Before you attempt to enable free-text search, you must set up free text search and the Guidewire Solr Extension, a full-text search engine, for external or embedded operation. After you complete the setup of free-text search, you enable free-text search in PolicyCenter with:

- The configuration parameter FreeTextSearchEnabled in config.xml
- The script parameter EnableDisplayBasicSearchTab in the running PolicyCenter application
- The free-text search plugins ISolrMessageTransportPlugin and ISolrSearchPlugin
- **Indexing System** rules in the **Event Fired** rule set
- The free-text message destination PCSolrMessageTransport.

None of the preceding free-text resources in PolicyCenter operate unless you set the turnkey configuration parameter FreeTextSearchEnabled to true. After you enable the preceding free-text resources, use the FreeTextSearchEnabled parameter to toggle the free-text search feature in PolicyCenter off and on temporarily.

To enable free-text search in PolicyCenter

1. Follow the instructions for “Free-text Search Setup” on page 89 in the *Installation Guide*.
2. Start PolicyCenter Studio.
3. In the Project window, navigate to **configuration** → **config** → **Rule Sets** → **EventMessage**.
4. Double-click **Event Fired** to open EventFired.grs in the Rules editor, and then enable the Indexing System rules.
5. In the Project window, navigate to **configuration** → **config** → **Plugins** → **registry**.
6. Double-click ISolrMessageTransportPlugin.gwp to open it in the Plugins Registry editor, and then enable the plugin with the Gosu class gw.solr.PCSolrMessageTransportPlugin.
7. In the Project window, navigate to **configuration** → **config** → **Plugins** → **registry**.
8. Double-click ISolrSearchPlugin.gwp and to open it in the Plugins Registry editor, and then enable the plugin with the Gosu class gw.solr.PCSolrSearchPlugin.
9. In the Project window, navigate to **configuration** → **config** → **Messaging**.
10. Double-click messaging-config.xml to open it in the Messaging editor.
11. In the list box on the left, select the SolrMessageTransport destination, and then mark the **Enabled** checkbox on the right.
12. In the Project window, navigate to **configuration** → **config**.
13. Double-click config.xml to open it in the XML editor, and then set FreeTextSearchEnabled to true.
14. Start the PolicyCenter application.

Result

If you enabled free-text search successfully, you see the following message at server startup on the server console and in the server log file.

```
***** PCSolrMessageTransportPlugin is initialized *****
```

If you configured free-text search for embedded operation, you see the following messages at server startup on the server console and in the server log file.

```
Solr Installing and provisioning embedded Solr in folder C:\opt\gwsolr  
Solr Embedded server configured for automatic provisioning in c:\opt\gwsolr for appCode pc
```

See also

- To set up free-text search for external or embedded operation, see “Free-text Search Setup” on page 89 in the *Installation Guide*.
- To learn more about the plugins and message destination, see “Free-text Search Integration” on page 593 in the *Integration Guide*

Configuring the Solr Extension for Integration with PolicyCenter

The following configuration files are important when you configure the Guidewire Solr Extension for free-text search.

- `pc-gwsolr.xml` – Defines the Guidewire Solr home directory for the Guidewire Solr Extension.
- `solr.xml` – Defines for the Guidewire Solr Extension the location in the Guidewire Solr home directory of each core for searchable entity types. For PolicyCenter, the only supported core is for policies.
- `solrserver-config.xml` – Configures how PolicyCenter connects to and works with the Guidewire Solr Extension. Categories of configuration settings include:
 - Connections with specific instances of the Guidewire Solr Extension
 - Type of operating mode for instances of the Guidewire Solr Extension
 - Provision of changed configuration files from the PolicyCenter home directory to instances of the Guidewire Solr home directory

Generally, you work with these files during installation, at the time you set up free-text search in the application server or servers dedicated to the Guidewire Solr Extension.

See also

- For details on the `pc-gwsolr.xml` and `solr.xml` configuration files, see “Free-text Search Setup” on page 89 in the *Installation Guide*.

Configuring Connections with the Guidewire Solr Extension

You configure instances of the Guidewire Solr Extension and how PolicyCenter connects with them in the `solrserver-config.xml` file. Two XML elements help in this type of configuration: the `<document>` element and the `<solrserver>` element.

The `<document>` Element

For each type of index document, such as policy data, a `<document>` element associates that data type with an instance of the Guidewire Solr Extension. For example:

```
<document name="policy" servername="my_solr_instance"/>
```

The `servername` attribute specifies an XML definition elsewhere in the `solrserver-config.xml` file.

The base configuration includes `<document>` elements for each type of data that free-text search supports. You must modify the `servername` attribute to match the instances of the Guidewire Solr Extension that you define and use.

The <solrserver> Element

For each instance of the Guidewire Solr Extension, a <solrserver> element defines its type of operation and how PolicyCenter connects with it.

```
<solrserver name="name" type={"embedded"|"http"|"cloud"}>
```

The `servername` attributes of <document> elements must match the name attributes of <solrserver> elements. You can map more than one <document> element to the same <solrserver> element.

The `type` attribute specifies the operating mode for the Guidewire Solr Extension. The attribute has three possible values:

- `embedded` – The Guidewire Solr Extension operates embedded within PolicyCenter.
- `http` – The Guidewire Solr Extension operates externally from PolicyCenter, as a single server.
- `cloud` – The Guidewire Solr Extension operates externally from PolicyCenter, as cluster of servers.

The base configuration includes <solrserver> elements that serve as examples for the <solrserver> elements you must define.

Configuring the Guidewire Solr Extension for Embedded or External Operation

You configure the Guidewire Solr Extension for embedded or external operation in the `solrserver-config.xml` file. The file contains one or more <solrserver> elements. They define instances of the Guidewire Solr Extension. You configure the Guidewire Solr Extension for embedded or external operation with the `type` attribute.

```
<solrserver name="name" type={"embedded"|"http"|"cloud"}>
```

Configuring the Guidewire Solr Extension for Embedded Operation

With embedded operation, the Guidewire Solr Extension runs as part of the PolicyCenter application, not as an application in a different application server instance. Therefore, embedded server definitions do not specify HTTP connection information.

The following example shows a typical configuration of an embedded server.

```
<solrserver name="embedded" type="embedded">
  <param name="solrroot" value="c:\opt\gwsolr"/>
</solrserver>
...
<document name="policy" archive="false" servername="embedded"/>
```

The `name` attribute lets you bind document types, such as policies, to a server that has cores for them. A <document> element must never reference a <solrserver> element of type `embedded` if you run the PolicyCenter application in production mode. If you do so, PolicyCenter generates an error message on the server console and in the server log, and free-text search does not operate.

For Solr servers of embedded type, you must specify the `solrroot` parameter. The value is the absolute path to a directory where the indexes for the cores are located. Generally, you specify a Guidewire Solr Home directory that holds files extracted from the `pc-gwsolr.zip` file as `solrroot`. In a typical development environment, the home directory is on the same host as the one that hosts your PolicyCenter application. Free-text search creates the directory specified by `solrroot` during server startup if the directory does not exist.

See also

- “Configuring the Guidewire Solr Extension for Provisioning” on page 349

Configuring the Guidewire Solr Extension for External Operation

With external operation, the Guidewire Solr Extension runs as an independent application in a different application server instance than the one that runs PolicyCenter. Therefore, external server definitions must specify HTTP connection information.

The following example shows a typical configuration of an external server for a development environment.

```
<solrserver name="localhost" type="http">
  <param name="host" value="localhost"/>
```

```
<param name="port" value="8983"/>
</solrserver>
...
<document name="policy" archive="false" servername="localhost"/>
```

The name attribute lets you bind document types, such as policies, to a server that has cores for them.

For external servers, you must specify the host and the port parameters. Typically in a development environment, you run the Guidewire Solr Extension in an application server instance hosted on the same machine where you run the PolicyCenter application. If you run the Guidewire Solr Extension on the same machine that runs PolicyCenter, specify localhost for the host parameter. Otherwise, specify the host name for the remote host.

In the default setup of the Guidewire Solr Extension, you configure its port number as 8983. For the port parameter of an external server definition, specify the port number that you configured for the Guidewire Solr Extension in the application server that runs it.

With external servers, you can specify two kinds of HTTP timeout parameters: connectiontimeout and readtimeout. The following example shows typical timeout parameter settings.

```
<solrserver name="localhost" type="http">
<param name="host" value="localhost"/>
<param name="port" value="8983"/>
<param name="connectiontimeout" value="300000"/>
<param name="readtimeout" value="300000"/>
<param name="maxConnections" value="1000000"/>
</solrserver>
```

Specify timeout intervals in milliseconds. The connectiontimeout parameter specifies how long PolicyCenter waits for the Guidewire Solr Extension to respond to a connection request. The readtimeout parameter specifies how long PolicyCenter waits for the Guidewire Solr Extension to completely return results from a search request.

See also

- “Configuring the Guidewire Solr Extension for High Availability” on page 350

Configuring the Guidewire Solr Extension for Provisioning

You can configure embedded server definitions to provision the Guidewire Solr Home directory with revised configuration files after you edit them in Studio. Use the provision parameter in an embedded server definition to control whether and how to provision the Guidewire Solr Home directory with changed configuration files.

```
<param name="provision" value="{"true"|"false"|"auto"}>
```

The following example shows a typical configuration of an embedded server with provisioning.

```
<solrserver name="embedded" type="embedded">
<param name="provision" value="true"
<param name="solrroot" value="c:\opt\gwsolr"/>
</solrserver>
```

If you set the provision parameter to true, free-text search deploys the files from PolicyCenter/gwsolr/pc-gwsolr.zip to the solrroot directory every time you start the PolicyCenter application. If you set provision to false, you must provision changed files that you edit in Studio. Set provision to auto only for automated testing. With provision set to auto, the indexes are dropped each time you start the PolicyCenter application.

To use provisioning successfully

- Always modify the free-text configuration files in Studio.
- Stop the PolicyCenter application, if it is running.
- Open a command prompt to PolicyCenter/bin and run the following command.

```
gwpc solr
```

The command rebuilds the pc-gwsolr.zip file in PolicyCenter/solr.
- Start the PolicyCenter application.

Free-text search deploys the contents of `pc-gwsolr.zip` to the directory specified by `solrroot`. Files already in `solrroot` are overwritten by new files from `pc-gwsolr.zip`.

See also

- “Configuring the Guidewire Solr Extension for Embedded Operation” on page 348

Configuring the Guidewire Solr Extension for High Availability

You configure the Guidewire Solr Extension for high availability by deploying it to multiple SolrCloud servers, managed as a cluster by Apache Zookeeper. Whenever you run the Guidewire Solr Extension with SolrCloud, you specify the host name and port number of the Zookeeper server in `solrserver-config.xml`. Use a `<solrserver>` element, and set the `type` attribute to “`cloud`”.

```
<solrserver name="cloud" type="cloud">
  <param name="host" value="zookeeperHostName"/>
  <param name="port" value="2181"/>
</solrserver>
```

See also

- For complete information on configuring a SolrCloud cluster in which to run the Guidewire Solr Extension, consult your Apache Solr documentation.

Configuring Free-text Search for Indexing and Searching

The following files configure how free-text search and the Guidewire Solr Extension operate together to index and search for policies.

- `schema.xml` – Defines search fields as known in the Guidewire Solr Extension.
- `policy-search-config.xml` – Defines the mapping between PolicyCenter fields and full-text search fields and configures how the full-text fields are indexed and searched.
- `data-config.xml` – Used by the Guidewire Solr Extension at startup to locate the Guidewire Solr home directory.

These files are pre-configured in the base configuration to index and search policy data. You do not need to modify these files unless you want to change the default set of search fields or their behaviors. The files also contain connection-related parameters that you set at the time you initially install and set up free-text search.

See also

- “Free-text Search Setup” on page 89 in the *Installation Guide*
- “Modifying Free-text Search for Additional Fields” on page 352

Configuring the Free-text Batch Load Command

The configuration and support files for the free-text batch load command and the command itself are located in the following directory on the host where the Guidewire Solr Extension resides.

```
opt/gwsolr/pc/solr/policy_active/conf
```

The following files configure the free-text batch load command.

- `batchload.sh/batchload.bat` – Specifies the batch load configuration file to use for your database brand.
- `batchload-config-databaseBrand.xml` – Defines the connection to the relational database that the free-text batch load command uses to query for policy data, as well as other system resources that the command requires. In addition, the batch load configuration file contains the native SQL Select statements that the batch load command uses to extract data from the PolicyCenter database.

Initial Configuration of the Free-text Batch Load Command

Generally, you configure the free-text batch load command when you first install and set up free-text search. Afterward, you need to modify your initial configuration only if resources in your computing environment change, such as the connection to your database.

For instructions on initial configuration, see “Setting Up the Free-text Batch Load Command” on page 99 in the *Installation Guide*.

SQL Select Statement Configuration for the Free-text Batch Load Command

The free-text batch load command extracts data from the PolicyCenter database by using native SQL. The native SQL Select statements that the batch loader uses are defined in configuration files for specific database brands. The configuration files that contain native SQL are:

- **For H2 databases** – batchload-config-h2.xml, suitable only for development
- **For Oracle databases** – batchload-config-oracle.xml, suitable for development or production
- **For SQL Server databases** – batchload-config-sqlserver.xml, suitable for development or production

You do not need to modify the native SQL when you first set up and configure the free-text batch load command. The native SQL correctly selects data for the search fields in the default configuration of free-text search. You must modify the native SQL only when you want to extend or modify the search fields for your configuration of free-text search.

For information on modifying the native SQL in the batch load configuration files, see “Modifying Free-text Search for Additional Fields” on page 352.

Configuring the Basic Search Screen for Free-text Search

The following table provides implementation details for each field on the **Search Policies → Basic** screen. The columns contain the following information:

- **Field** – The field on the **Search Policies → Basic** screen.
- **Entity** – The entity type of the object that corresponds to this field.
- **Search type** – Exact or inexact depending upon the search type.
- **Revised** – The marked cells indicate a revised field in which the contents can vary over time.

Use this information to help configure the **Basic** screen for free-text policy search.

Field	Entity	Search type	Revised
Policy Number	PolicyPeriod	Inexact	●
Name	PolicyContactRole subtype of: • PolicyPriNamedInsured • PolicyAddlNamedInsured	Inexact	
Street	PolicyAddress	Inexact	●
City	PolicyAddress	Inexact	●
State	PolicyAddress	Exact	●
Postal Code	PolicyAddress	Exact	●
Phone	Accessing the Contact through the PolicyContactRole.ContactDenorm field, get the phone number from the following fields: • HomePhone • PrimaryPhone • FaxPhone • WorkPhone	Exact	

Field	Entity	Search type	Revised
ID Number	Accessing the Contact through the PolicyContactRole.ContactDenorm field, get the ID number from the following fields: <ul style="list-style-type: none">• FEINOfficialID• SSNOfficialID	Exact	
Underwriting Company	PolicyPeriod	Exact	
Product	Policy	Exact	
Jurisdiction	PolicyPeriod	Exact	●
Producer of Record	PolicyPeriod.ProducerOfRecord	Exact	
Producer Code	Policy.ProducerCodeOfService	Exact	●
In Force On Date	Across multiple PolicyPeriod objects	Exact	

Limits on the Number of Free-text Search Results

You can limit the number of free-text search results returned from the Guidewire Solr Extension in two ways:

- Limit the number of free-text search results returned
- Set the number of free-text search results per page

Limiting the Number of Free-text Search Results Returned

The ISolrSearchPlugin plugin implementation limits the number of results that the Guidewire Solr Extension returns to PolicyCenter. You may want to increase or decrease the default value of 100 result items. Change the default by editing the ISolrSearchPlugin plugin registry to change the fetchSize parameter.

See also

- “Free-text Search Plugin” on page 596 in the *Integration Guide*

Setting the Number of Free-Text Results per Page

In the default configuration for free-text search, the basic search displays ten search results per page. You can configure the number of search results per page by modifying the pageSize property on the page configuration file. Regardless the number of results that you configure, users can change the page size to suit their needs after PolicyCenter displays the first page of results.

For basic policy search, modify the SolrPolicySearchPanelSet page configuration file. View and edit the file in the Project window in Studio by navigating to Page Configuration (PCF) → search → SolrPolicySearchPanelSet.

Modifying Free-text Search for Additional Fields

You can modify the configuration of free-text search in many ways. For example, you can add or remove fields for search criteria, modify how fields are stored in the Guidewire Solr Extension, and configure how fields are matched to search criteria. For complete information on how to modify the Guidewire Solr extension, consult the online documentation for Apache Solr 4.

This section shows by example the configuration files you typically modify to change how the Guidewire Solr Extension loads, indexes, and searches data. The example is a simple configuration change to add a field to free-text search.

IMPORTANT Be aware that adding multi-valued fields can affect free-text search performance. In particular, adding fields with too many values significantly degrades full-text search performance.

Configuration Files for Full-text Loading, Indexing, and Searching

Typically, you modify the following files to configure the way the Guidewire Solr extension loads, indexes, and searches information in the Guidewire Solr Extension. The following table lists the configuration files, grouped by the component they configure.

Configuration file	Description	
PolicyCenter		
policy-search-config.xml	Defines the mapping between PolicyCenter fields and Guidewire Solr Extension, and configures how Guidewire Solr Extension indexes and searches its index documents. PolicyCenter/modules/configuration/config/search/policy-search-config.xml	
PCSolrSearchPlugin.gs	The implementation class for the free-text plugin ISolrSearchPlugin. This plugin sends search criteria to the Guidewire Solr Extension and receives the search results. PolicyCenter/modules/pc/gsrc/gw/solr/PCSolrSearchPlugin.gs	
Guidewire Solr Extension		
schema.xml	Defines the document schema structure for the Guidewire Solr Extension. /opt/gwsolr/pc/solr/policy_active/conf/schema.xml	http://wiki.apache.org/solr/SchemaXml
data-config.xml	Read only at startup, defines where to locate and how to interpret data from the batch load command. /opt/gwsolr/pc/solr/policy_active/conf/data-config.xml	http://wiki.apache.org/solr/DataImportHandler#Configuration_in_data-config.xml-1
Free-text batch load command		
batchload.sh	The batch load command itself. /opt/gwsolr/pc/solr/policy_active/conf/batchload.sh	
batchload-config-database <i>Brand</i> .xml	Contains database connection information and brand-specific native SQL for selecting data from the PolicyCenter relational database. /opt/gwsolr/pc/solr/policy_active/conf/batchload-config.xml	
postprocess.sh	Collates and compiles index documents for the Guidewire Solr Extension using data selected from the relational database. /opt/gwsolr/pc/solr/policy_active/conf/postprocess.sh	

Sequence of Steps for Adding a Field to Free-text Search

Follow these high-level steps to configure free-text search with an additional field.

- “Define a New Free-text Field in the Guidewire Solr Extension” on page 353
- “Define a New Free-text Field in PolicyCenter” on page 354
- “Define a New Free-text Field in the Batch Load Command” on page 355

The examples that follow of add a policy postal code as a free-text field.

Define a New Free-text Field in the Guidewire Solr Extension

Open the following file to define a new free-text field in the Guidewire Solr Extension.

/opt/gwsolr/pc/solr/policy_active/conf/schema.xml

The Guidewire Solr Extension defines the format of this file.

The schema configuration file contains <field> elements, one for each full-text field of the index documents in the Guidewire Solr Extension.

The following example defines a full-text field that stores postal codes.

```
<field name="postalCode" type="gw_unanalyzed" indexed="true"
      stored="true" required="false"
      multiValued="false"/>
```

The definition directs the Guidewire Solr Extension to index the values of postal code fields, so search criteria can include postal codes. The definition directs the Guidewire Solr Extension to store the values of postal code fields, so items returned in search results include them.

Define a New Free-text Field in PolicyCenter

Open the following file to define a new free-text field in PolicyCenter.

`PolicyCenter/modules/configuration/config/search/policy-search-config.xml`

Access the file in the Project window in Studio by navigating to **configuration** → **config** → **search** → **policy-search-config.xml**. PolicyCenter defines the format of this file.

Free-text search configuration files have these main elements:

- **<Indexer>** – Contains **<IndexField>** elements to define field names and their locations within object graphs of the root object and in the index documents sent to the Guidewire Solr Extension.
- **<Query>** – Contains the following types of elements:
 - **<FilterTerm>** elements configure whether to return specific fields in results if the field matches search criteria.
 - **<QueryTerm>** elements define how specific fields are matched and how a match contributes to the overall score. A query term is one of two types: *term* and *subquery*. A term type searches a single index. A subquery type searches multiple indices simultaneously and scores the most appropriate match.
- **<QueryResult>** – Contains **<ResultProperty>** elements to configure whether and how specific fields are returned in query results from the Guidewire Solr Extension.

To add a free-text field to `policy-search-config.xml`, first add an **<IndexField>** element to the **<Indexer>** element. The following example defines a free-text field for postal codes.

```
<IndexField field="postalCode">
  <DataProperty path="root.PolicyAddress.PostalCode"/>
</IndexField>
```

The definition specifies that values of `postalCode` fields in the index documents sent to the full-text engine come from addresses on policy periods, the root object.

Next, add **<FilterTerm>** elements for the new free-text field to the **<Query>** element. The following example specifies how the Guidewire Solr Extension matches `PostalCodeCriteria` values in search criteria with `postalCode` values in the Guidewire Solr Extension.

```
<FilterTerm>
  <DataProperty path="root.PostalCodeCriteria"/>
  <QueryField field="postalCode"/>
</FilterTerm>
```

The definition directs the Guidewire Solr Extension to accept postal codes in search criteria and where to find them in the XML structure of the search criteria.

Finally, add a **<ResultProperty>** element to the **<QueryResult>** element. The following example defines how the Guidewire Solr Extension returns `postalCode` values in query results.

```
<ResultProperty name="PostalCode">
  <ResultField name="postalCode"/>
</ResultProperty>
```

The definition assigns the `postalCode` value in the result to the `PostalCode` on the result object.

Define a New Free-text Field in the Batch Load Command

In order to load data from an existing PolicyCenter database, the new free-text field has must listed in the `batchload-config.xml` files and the `data-config.xml` file. You decide whether to include or exclude the field from the digest that prevents duplicate index entries.

Generating the SQL Query

Add the new free-text field in the SELECT portion of the query that corresponds to the data. The SQL for postal codes looks like following sample SELECT statement.

```
SELECT DISTINCT
...
    paddr.postalcodeinternal AS postalCode
FROM pc_policyperiod AS pp
...
    INNER JOIN pc_policyaddress paddr
        ON paddr.branchid = pp.id
...
WHERE
...
    AND (paddr.EffectiveDate IS NULL OR paddr.EffectiveDate <= pp2.EditEffectiveDate )
    AND (paddr.ExpirationDate IS NULL OR paddr.ExpirationDate > pp2.EditEffectiveDate )
...
```

This is duplicated in `batchload-config-oracle.xml`, `batchload-config-sqlserver.xml`, and `batchload-config-h2.xml`. There is one file per database brand because requires the query to be written a little differently.

Note: Because the SQL is included in an XML file, you must escape the less than (`<`) and greater than (`>`) symbols.

The query is processed into an XML document that will be loaded into SQL. Batch loading of XML into Solr is described in http://wiki.apache.org/solr/DataImportHandler#Configuration_in_data-config.xml-1.

The field names come out all in uppercase, and the structure of the XML document is

```
<CONTAINER_ELEM>
<POLICY>
    <!-- data for one policy -->
</POLICY>
</CONTAINER_ELEM>
```

The entry needed in `data-config.xml` in order to include the `postalCode` in the index looks like the following XML code.

```
<field column="postalCode" xpath="/CONTAINER_ELEM/POLICY/POSTALCODE"/>
```

Within the row for one claim contact, the fields will be in the same order as they were returned by the SELECT statement.

You can have fields in the SQL result that do not become part of the XML of the index documents. The Guidewire Solr Extension ignores them when it loads them. The batch load command uses these kinds of fields to sort and manipulate the data returned from the database to XML to produce the final XML index documents to load. But, these fields are not part of the index document schema.

Computing the Hash to Eliminate Duplicates

To avoid a chain of policy changes or renewals generating identical, duplicate index entries, we make a SHA-1 hash of most of the index data. It is included in the URN (Unique Record Name) of the index entry. Entries on the same policy and with the same SHA hash are collapsed into a single entry.

First decide whether the field is part of the hash. If so, we must modify the file `PCSolrSearchPlugin.gs` to include that data in the digest when generating a new index entry. The free-text batch load command includes fields by default.

One line of Gosu is required.

```
sb.append("postalCode", period.PolicyAddress.PostalCode)
```

If on the other hand, this field is *not* part of the hash, then we need to tell the batch load command to ignore it for digest purposes. An example of such a field is `sliceDate`. Near the bottom of the `batchload-config.xml` files is the configuration for the digester. You can see that `sliceDate` is listed among the columns that are ignored.

```
<transformer  
    name="digestTransformer"  
    class="com.guidewire.solr.batchload.xform.PCDigestTransformer"  
    ignoreElems="urn, periodID, policyPublicID, sliceDate, periodStart, periodEnd, policyStart,  
                policyEnd, periodIdWithSliceDate, jobType"  
    algorithm="SHA"  
/>
```

For default columns, include this in `PCSolrSearchPlugin.gs` as well.

```
static function initXformer() : com.guidewire.solr.batchload.xform.DigestTransformer {  
    try {  
        var xml =  
            "<transformer name=\"digestTransformer\""  
            "class=\"com.guidewire.solr.batchload.xform.PCDigestTransformer\""  
            "+ " algorithm=\"SHA\""  
            "+ " ignoreElems=\"urn, periodID, policyPublicID, sliceDate, periodStart, periodEnd, ""  
            "+ " policyStart, policyEnd, periodIdWithSliceDate, jobType\"/>";  
        var xf = new com.guidewire.solr.batchload.xform.PCDigestTransformer(false)  
        var doc = com.guidewire.solr.batchload.Utils.parseXml(xml)  
        xf.configure(doc.getDocumentElement())  
    }
```

There is a final stage to the de-duplication. The `postprocess` script sorts the data by a combination of urn, slice date, and term number. It then removes rows with duplicate urn values. Slice date and term number are used in the initial sort to retain the latest version of the duplicate data, which is the same way that PolicyCenter indexes the data.

Configuring Special Page Functions

This topic describes how to configure special functionality related to pages.

This topic includes:

- “Adding Print Capabilities” on page 357
- “Adding Print Capabilities” on page 357
- “Linking to a Specific Page: Using an EntryPoint PCF” on page 359
- “Linking to a Specific Page: Using an ExitPoint PCF” on page 361

Note: The code samples included in this topic assume that you are using the ClaimCenter application. Any listed data model objects or fields are specific to that application. However, the features documented in this topic are universal to all Guidewire applications.

Adding Print Capabilities

You can customize the print functions on the PolicyCenter interface. This section explains the print capabilities and how to use them. It covers the following topics:

- Overview of the Print Functionality
- List View Printing

Overview of the Print Functionality

You can use the PolicyCenter print functionality to print the list view parts of the data visible on a PolicyCenter screen. You can control both the output format and which list view objects are printed. Most commonly, pages print as PDF but PolicyCenter also supports a limited comma-separated values (CSV) format. You can send the output of a print action to a local printer or save it to disk. From PolicyCenter, you can print, for example, object lists such as the **Activities** list on the **Desktop**.

All client machines must have a supported version of the Acrobat Reader available to support printing.

Configuration Parameters Related to Printing

The following optional print parameters in `config.xml` control the default print settings globally. For information on configuration parameters, see “Application Configuration Parameters” on page 35.

<code>DefaultContentDispositionMode</code>	Specifies the Content-Disposition setting to use if the content to be printed is returned to the browser. Must be either “attachment” (the default) or “inline”.
<code>PrintFontFamilyName</code>	Sets the name of font family to use for output. The default is “sans-serif”.
<code>PrintFontSize</code>	Sets the page font size. The default is 10 points.
<code>PrintFOPUserConfigFile</code>	Sets the fully qualified path to a valid FOP user configuration file. Use this to specify or override the default FOP configuration.
<code>PrintHeaderFontSize</code>	Sets the header’s font size. The default is 16 points.
<code>PrintLineHeight</code>	Specifies the line height. The default is 14 points.
<code>PrintListViewBlockSize</code>	Sets the block size of the list elements if printing a list with elements.
<code>PrintListViewFontSize</code>	Sets the font size for printing list views. The default is 10 points.
<code>PrintMarginBottom</code>	Sets the bottom margin. The default is .5 inches.
<code>PrintMarginLeft</code>	Specifies the size of left margin. The default is 1 inch.
<code>PrintMarginRight</code>	Specifies the size of right margin. The default is 1 inch.
<code>PrintMarginTop</code>	Specifies the size of top margin. The default is .5 inches.
<code>PrintMaxPDFInputFileSize</code>	Sets the size of the intermediate XML file to create if printing to PDF.
<code>PrintPageHeight</code>	Specifies the height of the page. The default is 8.5 inches.
<code>PrintPageWidth</code>	Specifies the width of the page. The default is 11 inches.

You can modify any of the page formatting attributes using property values defined in the CSS2 specification. The specification resides online at the following location:

<http://www.w3.org/TR/REC-CSS2>

Security Related to Printing

PolicyCenter provides a `lvprint` system permission that gives you the ability to print the information that appears in a list view. The following table lists the base configuration roles with the `lvprint` permission.

Roles with `lvprint` permission

- Superuser
- Integration Admin

Gosu API Methods for Printing

PolicyCenter contains a Gosu API that contains a number of print-related methods. It is:

```
gw.api.print
```

Guidewire recommends that you avoid using this API in your print configurations with the exception of the following methods:

- `ListViewPrintOptionsPopupAction`
- `PrintSettings`

List View Printing

To add print capabilities to a PCF page you must decide which type of printing you require. You can configure the following types of printing behaviors in PolicyCenter.

- *List view printing* that you use to output a list in PDF or CSV format.

List View Printing

You use list view printing to output a list in either a PDF format or a CSV (comma-separated value) format. You can interactively choose the type of output delivered by the Print button. This type of printing uses the `ToolbarButton` element with an `action` attribute. The `action` attribute contains a Gosu expression that calls the Gosu API `ListViewPrintOptionsPopupAction.printListViewWithOptions("MyListView")`. The following example illustrates a list view printing method:

```
<ToolbarButton label="displaykey.Java.ListView.Print" id="PrintButton"
    action="gw.api.print.ListViewPrintOptionPopupAction.printListViewWithOptions('MyListView')"/>
```

If you choose to print in CSV format, you can also choose which columns to print. For example, you can use list view printing to print the [Desktop Activities](#) page.

Linking to a Specific Page: Using an EntryPoint PCF

It is possible to connect directly to PolicyCenter using a URL that leads to a specific PolicyCenter page. You can define your own links or entry points. Thus, if the PolicyCenter server receives a connect request from an external source and the request has both the correct format and parameters, PolicyCenter serves the requested page.

In the base configuration, PolicyCenter provides a number of `EntryPoint` PCF examples. You can find these in the following location in Studio:

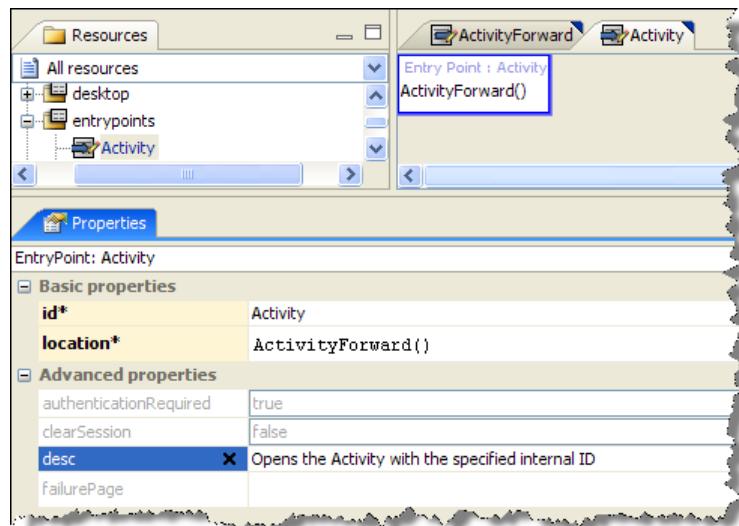
`configuration → config → Page Configuration → pcf → entrypoints`

These PCF pages are examples only. If you use one, you must customize it to meet your business needs. You can also use them as starting points for your own `EntryPoint` PCF pages.

Entry Points

An entry point takes the form of a URL with a specific syntax. The entry URL specifies a location that a user enters into the browser. If the PolicyCenter server receives a connection request with a specific entry point, PolicyCenter responds by serving the page based on the entry point configuration.

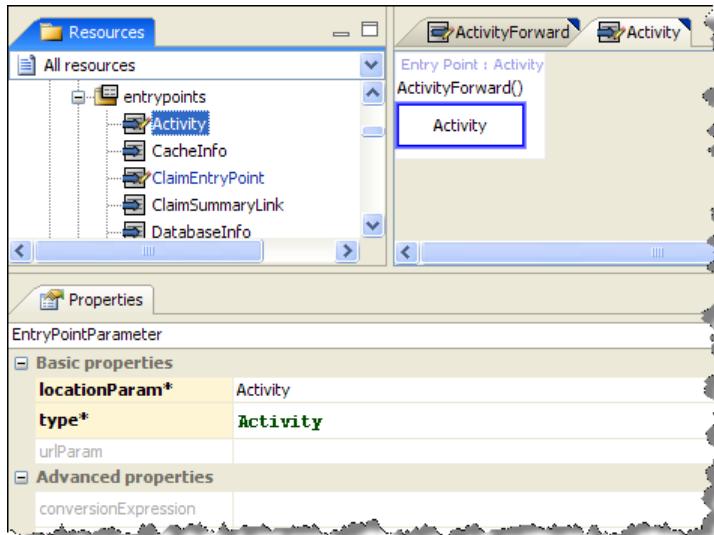
To implement this functionality, you must create an `EntryPoint` PCF (in the `entrypoints` folder). The following graphic illustrates an `EntryPoint` PCF.



The EntryPoint PCF contains the following parameters:

<code>authenticationRequired</code>	Specifies that PolicyCenter must authenticate the user before the user can access the URL. If true, PolicyCenter requires that the user already be authenticated to enter. If the user is not already logged in, PolicyCenter presents a login page before rendering the entry point location. The default is true. Guidewire strongly recommends that you think carefully before setting this value to false.
<code>clearSession</code>	If true, clears the server session for this user as the user enters this entry point
<code>desc</code>	Currently, does nothing.
<code>failurePage</code>	Specifies the page to send the user if PolicyCenter can not display the entry point. Failures typically happen any time that the data specified by the URL does not exist. The default is Error.
<code>id</code>	Required. The PolicyCenter uniform resource identifier to show, minus its .do suffix. Typically, this is the same as the page ID. No two EntryPoints can use the same URI. Do not use the main application name, PolicyCenter, as the URI. For example, if the URI is XXX, then it is possible to enter the application at <code>http://myserver/myapp/XXX.do</code> .
<code>location</code>	Required. The ID of the page, Forward, or wizard to which you want to go. Guidewire recommends that if you want the entry point to perform complex logic, use a Forward. See "To create a forwarding EntryPoint PCF" on page 361 for a definition of a forward.

Each EntryPoint PCF can contain one or more EntryPointParameter subelements that specifies additional functionality.



The `EntryPointParameter` subelement has the following attributes:

<code>conversionExpression</code>	Gosu expression that PolicyCenter uses to convert a URL parameter to the value passed to the location parameter.
<code>desc</code>	Currently, does nothing.
<code>locationParam</code>	Required. The name of the LocationParameter on the EntryPoint target location that this parameter sets.
<code>optional</code>	Specifies whether the parameter is optional. If set to true, PolicyCenter does not require this parameter.
<code>type</code>	Required. Specifies what type to cast the incoming parameter into, such as String or Integer.
<code>urlParam</code>	The name of the parameter passed with the URL. For example, if the urlParam is Activity and the entry point URI is ActivityDetail, you would pass Activity 3 as: <code>http://myserver/myapp/ActivityDetail.do?Activity=3</code>

Creating a Forwarding EntryPoint PCF

A *forward* is a top-level PCF location element similar to a page or wizard. However, it has no screen. It merely forwards you to another location. You define a forward separately from the **EntryPoint** PCF. However, you set the forward for a PCF in the **EntryPoint** PCF location attribute.

Note: For an example of how to define a forward, see **ActivityForward** in PolicyCenter Studio at `pcf → activity → ActivityForward`.

To create a forwarding EntryPoint PCF

1. Define a separate entry point (PCF) with `authenticationRequired` property set to `false`. This PCF is effectively a forwarding page to handle the seamless login.
2. Set the `location` attribute of the entry point to use a `Forward` to call the `AuthenticationServicePlugin`.
3. Do one of the following:
 - If the plugin login is successful, forward the user onto the actual page (the desktop, for example) to which you intended to send the user in the first place. (This is the page to which the user would have gone if `authenticationRequired` had been set to `true`.)
 - If the plugin login is not successful, redirect the user to an error page or an alternate login page.

Suppose that there are several destinations to which you wish the user to go. In this case, consider passing a parameter to the entry point forward, so you can have the seamless login logic all in that one place.

Linking to a Specific Page: Using an ExitPoint PCF

It is possible to create a link from a PolicyCenter application screen to a specific URL. This URL can be any of the following:

- A page in another Guidewire application
- A URL external to the Guidewire application suite

You provide this functionality by creating an **ExitPoint** PCF file and then using that functionality in a PolicyCenter screen.

In the base configuration, PolicyCenter provides a number of **ExitPoint** PCF examples. You can find these in the following location in Studio:

`configuration → config → Page Configuration → pcf → exitpoints`

Note: These PCF pages are examples only. If you use one, Guidewire expects you to customize it to meet your business needs. You can also use them as starting points for your own **ExitPoint** PCF pages.

Creating an ExitPoint PCF

The following example takes you through the process of creating a new exit point PCF and then modifying a PolicyCenter interface screen to use the exit point. It does the following:

- Step 1 creates a new **ExitPoint** PCF page with the required parameters.
- Step 2 modifies the **Activity Detail** screen by adding a new **Dynamic URL** button. If you click this button, it opens a new popup window and loads the Guidewire Internet home page into it.
- Step 3 tests your work and verifies that the button works as intended.

It is possible to use any action attribute to activate the **ExitPoint** PCF. This example uses a button input as it is the easiest to configure and test. This example pushes the URL to a popup window that leaves the user logged into PolicyCenter. You can also configure the **ExitPoint** PCF functionality to log out the user or to possibly reuse the current window.

Step 1: Create the ExitPoint PCF File.

The first step is to create a new **ExitPoint** PCF file and name it AnyURL.

1. Within Studio, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **exitpoints**, and then select **New** → **PCF File** from the right-click menu.
2. Enter AnyURL for the file name in the **New PCF File** dialog and select **Exit Point** as the file type.
3. Select the **AnyURL** file, so that Studio outlines the **ExitPoint** element in blue.
4. Select the **Properties** tab at the bottom of the screen and set the listed properties. This example pushes the URL to a popup window that leaves the user logged into PolicyCenter. You can also configure the **ExitPoint** PCF functionality to log out the user or to possibly reuse the current window.
 - **logout** — **false**
 - **popup** — **true**
 - **url** — **{exitUrl}**
5. Select the **Entry Points** tab and add the following entry point signature:
`AnyURL(url : String)`
6. In the **Toolbox**, expand the **Special Navigation** node, select the **Exit Point Parameter** widget, and drag it into your exit point PCF.
7. Select the **Exit Point Parameter** widget and enter the following in its **Properties** tab:
 - **locationParam** — **url**
 - **type** — **String**
 - **urlParam** — **exitUrl**

Step 2: Modify the User Interface Screen to Use the Exit Point

After you create the **ExitPoint** PCF, you need to link its functionality to a PolicyCenter screen. The **Activity Detail** screen contains a set of buttons across the top of the screen. This example adds another button to this set of buttons. It is this button that activates the exit point.

1. In Studio, create a new **Button.Activity.DynamicURL** display key. You need this display key as a label for the button that you create in a later step.
 - a. Open the **Display Key** editor and navigate to **Button** → **Activity**.
 - b. Select the **Activity** node, right-click and select **Add**.
 - c. Enter the following in the **Display Key Name** dialog:
 - **Display Key Name** — **Button.Activity.DynamicURL**
 - **Default Value** — **Dynamic URL**
2. Open the PCF for the page on which you want to add the exit point. For the purposes of this example, open the **ActivityDetailScreen** PCF file.

Note: The simplest way to find a Studio resource is to press **CTRL+N** and enter the resource name.
3. Select the entire **ActivityDetailScreen** element on the PCF page. Studio displays a blue border around the selected element.
4. In the **Code** tab at the bottom of the screen, enter the following as a new function:

```
// This function must return a valid URL string.
function constructMyURL() : String { return "http://www.guidewire.com" }
```

You can make the actual function as complex as you need it to be. The function can also accept input parameters as well. The only stipulation is that it must return a valid URL string.

5. In the **Toolbox** for the PCF page that you just opened, find a **Toolbar Button** widget and drag it into the line of buttons at the top of the page.
6. Select the new button widget so that it has a blue border around it.
7. Select the **Properties** tab at the bottom of the screen and set the listed properties. It is possible to use any action attribute to activate the **ExitPoint** PCF. This example uses a button input as it is the easiest to configure and test.
 - **action** — AnyURL.push(constructMyURL())
 - **id** — DynamicURL
 - **label** — displaykey.Button.Activity.DynamicURL

Step 3: Test Your Work

After completing the previous steps, you need to test that the button you added to the **Activity Detail** screen works as you intended.

1. Start the PolicyCenter application server, if it is not already running. It is not necessary to restart the application server as you simply made changes to PCF files. You did not actually make any changes to the underlying PolicyCenter data model, which would require a server restart.
2. Log into PolicyCenter using an administrative account.
3. Press ALT+SHIFT+T to open the **Server Tools** screen. This screen is only available to administrative accounts.
4. Choose **Reload PCF Files** in the **Internal Tools** → **Reload** screen. PolicyCenter presents a success message after it reloads the PCF files from the local file system.
5. Log into PolicyCenter under a standard user account and search for an activity. The **Activity Detail** screen now contains a **Dynamic URL** button.
6. Click the **Dynamic URL** button and PolicyCenter opens a popup window and loads the URL that you set on the **constructMyURL** function. If you followed the steps of this example exactly, PolicyCenter loads the Guidewire Internet home page into the popup window.

Workflow and Activity Configuration

Using the Workflow Editor

This topic covers basic information about the workflow editor in Guidewire Studio.

This topic includes:

- “Workflow in Guidewire PolicyCenter” on page 367
- “Workflow in Guidewire Studio” on page 368
- “Understanding Workflow Steps” on page 369
- “Using the Workflow Right-Click Menu” on page 370
- “Using Search with Workflow” on page 370

Workflow in Guidewire PolicyCenter

Guidewire PolicyCenter uses workflow to drive various key business processes (Renewals, Policy Changes, Submissions, and similar items). Guidewire defines and stores each base configuration workflow process as a separate file in the following directory:

```
modules/configuration/config/workflow
```

Each file name corresponds to the workflow process that it defines (for example, `CompleteCancellationWF.1.xml`). Each workflow file name contains a version number. If you create a new workflow, Studio creates a workflow file with version number 1. If you modify an existing base configuration workflow, Studio creates a copy of the file and increments the version number. In each case, Studio places the workflow file in the following directory:

```
modules/configuration/config/workflow
```

See also

- For a discussion of workflow as it relates to specific job types (Submissions, for example), see “Policy Transactions” on page 73 in the *Application Guide*.
- For information on workflow structure and design, see “Guidewire Workflow” on page 373.

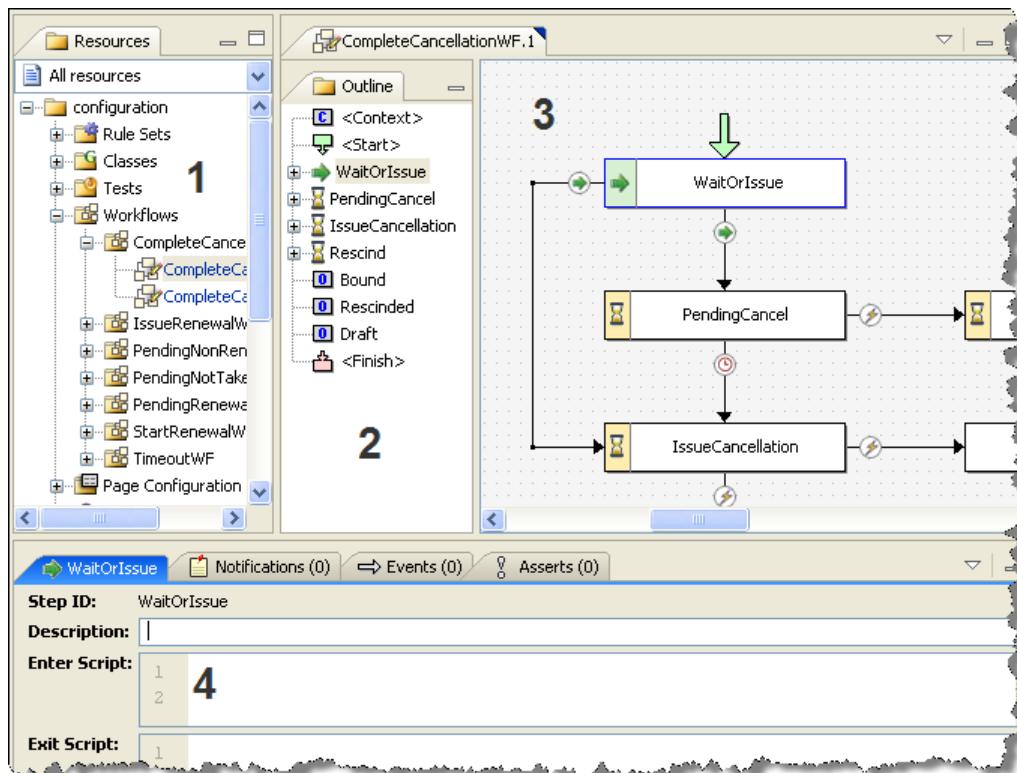
Workflow in Guidewire Studio

Even though Guidewire defines the workflow scripts in XML files, you use Guidewire Studio to view, edit, manage, and create new workflows scripts. Thus, you do not work directly with XML files. Instead, you work with their representation in Guidewire Studio, in the Studio **Workflows** editor.

To access the workflow editor, navigate to **configuration** → **config** → **Workflows**, and then select a workflow. Within the **Workflows** editor, there are multiple work areas, each of which performs a specialized function:

Area	View	Description
1	Tree view	Studio displays each workflow type as a node in the Resources tree. If you have multiple versions of a workflow type, Studio displays each one with an incremental version number at the end of the file name.
2	Outline view	Studio displays an outline of the selected workflow process in the Outline pane. This outline lists all the steps and branches for the workflow in the order that they actually appear in the workflow XML file. You can re-order these steps as desired. You can also re-order the branches within a step. First, select an item, then right-click and select the appropriate menu item.
3	Layout view	Studio displays a graphical representation of the workflow in the workflow pane. You use this representation to visualize the workflow. You also use it to edit the defining values for each step and branch.
4	Property view	Studio displays detailed properties for the selected step or branch, much of which you can modify.

For example, in the PolicyCenter base configuration, Guidewire defines a **CompleteCancellationWF** script. In Studio, it looks similar to the following:



The following table lists the main workflow elements and describes each one.

Element	Editor	Description	See...
<Context>		Every workflow begins with a <Context> block. You use it to conveniently define symbols that apply to the workflow.	"<Context>" on page 379
<Start>		Defines the step on which the workflow starts. It optionally contains Gosu blocks to set up the workflow or its business data. It runs before any other workflow step.	"<Start>" on page 379
AutoStep		Defines a workflow step that finishes immediately, without waiting for time to pass or for an external trigger to activate it.	"AutoStep" on page 381
MessageStep		Supports messaging-based integrations. It automatically generates and sends a single integration message and then stops the workflow until the message completes. (Typically, this is through receipt of an ack return message.) After the message completes, the workflow resumes automatically.	"MessageStep" on page 382
ActivityStep		An ActivityStep is similar to an AutoStep, except that it can use any of the branch types, such as a TRIGGER or a TIMEOUT, to move to the next step. However, before an ActivityStep branches to the next step, it waits for one or more activities to complete.	"ActivityStep" on page 383
ManualStep		Defines a workflow step that waits for someone—or something—to invoke an external trigger or for some period of time to pass.	"ManualStep" on page 384
GO		Indicates a branch or transition to another workflow step. It occurs only within an AutoStep workflow step. <ul style="list-style-type: none"> • If there is only a single GO element within the workflow step, branching occurs immediately upon workflow reaching that point. • If there are multiple GO elements within the workflow step, each GO element (except the last one) must contain conditional logic. The workflow then determines the appropriate next step based on the defined conditions. 	"GO" on page 387
TRIGGER		Indicates a branch or transition to another workflow element. It occurs only within a ManualStep workflow step. Branching occurs only upon manual invocation from outside the workflow.	"TRIGGER" on page 388
TIMEOUT		Indicates a branch or transition to another workflow element. It occurs only within a ManualStep workflow step. Branching to another workflow step occurs only after a specific time interval has passed.	"TIMEOUT" on page 390
Outcome		Indicates a possible outcome for the workflow. This step is special. It indicates that it is a last step, out of which no branch leaves.	"Outcome" on page 385
<Finish>		(Optional) Defines a Gosu script to run at the completion of the workflow to perform any last clean up after the workflow reaches an outcome. It runs after all other workflow steps.	"<Finish>" on page 379

Understanding Workflow Steps

Each workflow step represents a location in the workflow. It does not have a business meaning outside of the workflow. Therefore, it is permissible to use whatever IDs you want and arrange them however it is most convenient for you. (Beware, however, of infinite cycles between steps. PolicyCenter treats too many repetitions between steps as an error.)

A workflow script can contain any of the following steps. It must contain at least one **Outcome** step. It must also start with one each of the <Context> and <Start> steps described in “Workflow Structural Elements” on page 378.

Type	Workflow contains	Icon	Step	Description
AutoStep	Zero, one, or more		Step1	Step that PolicyCenter guarantees to finish immediately. See “AutoStep” on page 381.
ManualStep	Zero, one, or more		Step2	Step that waits for an external TRIGGER to occur or a TIMEOUT to pass. See “ManualStep” on page 384.
ActivityStep	Zero, one, or more		Step3	Step that waits for one or more activities to complete before continuing. See “ActivityStep” on page 383.
MessageStep	Zero, one, or more		Step4	Special-purpose step designed to support messaging-based integrations. See “MessageStep” on page 382.
Outcome	One or more		Outcome	Special final step that has no branches leading out of it. See “Outcome” on page 385.

Using the Workflow Right-Click Menu

You can modify a workflow step by first selecting it, then selecting different items from the right-click menu.

Desired result	Actions
To change a workflow step name	Select Rename from the right-click menu. This opens the Rename StepID dialog in which you can enter the new step name.
To change a workflow step type	Select Change Step Type from the right-click menu, then the type of workflow step from the submenu. This action opens a dialog in which you set the new workflow step type parameters.
To move a workflow step up or down	Select Move Up (Move Down) from the right-click menu. The editor only presents valid choices for you to select. This action moves the workflow step up or down within the workflow outline view.
To create a new branch	Select New <BranchType> from the right click menu. The editor presents you with valid branch types for the workflow step type. This action opens a dialog in which you set the new branch parameters.
To delete a workflow step	Select Delete from the right-click menu. This action removes the workflow step from the workflow outline. The workflow editor does not permit you to remove the workflow step that you designate as the workflow start step.

See also

- To learn how to localize names of workflow steps, see “Localizing Guidewire Workflow” on page 65 in the *Globalization Guide*.

Using Search with Workflow

It is possible to search for a specific text string within a workflow by selecting **Find in Path** from the Studio **Edit** menu. You can search on a localized text strings as well. You can also select a workflow and select **Find in Path** from the right-click menu.

- If you use the **Search** menu option, you can filter the resources to check.
- If you use the right-click menu option, then the search encompasses all active resources.

In either case, Studio opens a search pane at the bottom of the screen and displays any matches that it finds. You can click on a match to open the workflow in which the match exists.

Guidewire Workflow

This topic covers PolicyCenter workflow. Workflow is the Guidewire generic component for executing custom business processes asynchronously.

This topic includes:

- “Understanding Workflow” on page 373
- “Workflow Structural Elements” on page 378
- “Common Step Elements” on page 379
- “Basic Workflow Steps” on page 381
- “Step Branches” on page 386
- “Creating New Workflows” on page 391
- “Instantiating a Workflow” on page 395
- “The Workflow Engine” on page 397
- “Workflow Subflows” on page 400
- “Workflow Administration” on page 401
- “Workflow Debugging, Logging, and Testing” on page 402

Understanding Workflow

There are multiple ways to think about workflow:

Term	Definition
workflow, workflow instance	A specific running instance of a particular business process. Guidewire persists a workflow instance to the database as an entity called <code>Workflow</code> .
workflow type	A single kind of flow process, for example, a Cancellation workflow.
workflow process	A definition of a workflow type in XML. Guidewire defines workflow processes in XML files that you manage in Guidewire Studio through the graphical <code>Workflows</code> editor.

Discussions about *workflow* in general or the *workflow system* refer usually to the workflow infrastructure as a whole.

Workflow Instances

Think of a *workflow instance* as a row in the database marking the existence of a single running business flow. PolicyCenter creates a workflow instance in response to a specific need to perform a task or function, usually asynchronously. For example, in the base configuration, PolicyCenter creates a *macro* workflow as a companion to each newly-created Job (Submission, PolicyChange, Renewal, for example). This workflow is responsible for pushing the Job through its life cycle.

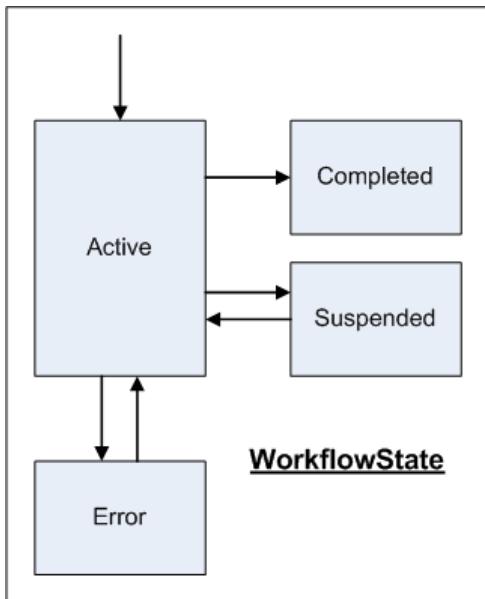
The newly created instance takes the form of a database entity called `Workflow`. (For more information on the `Workflow` entity, consult the PolicyCenter *Data Dictionary*.) Because PolicyCenter creates the `Workflow` entity in a bundle with other changes to its associated business data, PolicyCenter does nothing with the workflow until it commits the workflow. PolicyCenter does not send messages to any external application unless the surrounding bundle commits successfully.

After creation of the `Workflow` entity, nothing further happens from the viewpoint of the code that created the workflow. The workflow merely continues to execute asynchronously, in the background, until it completes. It is not possible, in code, to wait on the workflow (as you can wait for a code thread to complete, for example). This is because some workflows can literally and deliberately take months to complete.

All workflows have a *state* field (a typekey of type `WorkflowState`) that tracks how the workflow is doing. This state—and the transitions between states—is extremely simple:

- All newly beginning `Workflow` entities start in the `Active` state, meaning they are still running.
- If a `Workflow` entity finishes normally, it moves to the `Completed` state, which is final. A workflow in the `Completed` state takes no further action, it exists from then on only as a record in the database.
- If you suspend a workflow, either from the PolicyCenter **Administration** interface, or from the command line, or through the Workflow API, the workflow moves to the `Suspended` state. A workflow in the `Suspended` state does nothing until manually resumed from the **Administration** interface, from the command line, or through the Workflow API.
- If an error occurs to a workflow executing in the background, the workflow moves into the `Error` state after it attempts the specified number of retries. A workflow in the `Error` state does nothing until manually resumed from the **Administration** interface, the command line, or the Workflow API.

The following graphic illustrates the possible workflow states:



Notice that this diagram does not convey any information about how an active workflow (a workflow in the Active state) is actually processing. For active workflows, Guidewire defines the workflow state in the `WorkflowActiveState` typelist, which contains the following states:

- Running
- WaitManual
- WaitActivity
- WaitMessage

Whether the workflow is actually running depends on whether it is the current *work item* being processed.

Work Items

Each running workflow instance can have a *work item*. (See “Work Queues” on page 100 in the *System Administration Guide* for more information on work items.) If a running workflow does not have a work item associated with it, the workflow writer picks up the workflow instance at the next scheduled run. The state of this work item is one of the following:

- Available
- Failed – PolicyCenter retries a Failed work item up to the maximum retry limit.
- Checkedout – PolicyCenter processes a Checkedout work item in a specific worker's queue after the work item reaches the head of that queue.

For the specifics of configuring work queues, see “Configuring Work Queues” on page 105 in the *System Administration Guide*.

Workflow Process Format

To structure a workflow script, Guidewire uses the concept of a directed graph that shows how the `Workflow` instance moves through the various states. (This is known formally as a Petri net or P/T net.) Guidewire calls each state a *Step* and calls a transition between two states a *Branch*. Guidewire defines multiple types of steps and branches.

Even though Guidewire defines the workflow scripts in XML files, you use Guidewire Studio to view, edit, manage, and create new workflows scripts. See “Workflow in Guidewire Studio” on page 368.

Workflow Step Summary

The workflow process consists of the following steps (or states). The table lists the steps in the approximate order in which they occur in the workflow script. A designation as *structural* indicates that these steps are mandatory and that Studio inserts them into the workflow process automatically. Studio marks the structural steps with brackets (< . . . >) to indicate that they are actually XML elements. Some of the structural elements have no visual representation within the workflow diagram itself. You can only choose them from the workflow outline.

Step	Script contains	Description
<Context>	Exactly one	Structural. Element for defining symbols used in the workflow. Generally, you define a symbol to use as convenience in defining objects in the workflow path. For example, you can define a symbol such that inserting “Cancellation” into the text workflow actually inserts “Workflow.PolicyPeriod.Cancellation”.
<Start>	Exactly one	Structural. Element defining on which step the Workflow element starts. It can optionally contain Gosu code to set up the workflow or its business data.
AutoStep ActivityStep ManualStep MessageStep	Zero, one, or more	<p>A step is one stage that the Workflow instance can be in at a time. There can be zero, one, or more of any of these steps, in any order.</p> <p>Each of these steps in turn can contain one or more of the following:</p> <ul style="list-style-type: none"> • Any number of Assert code blocks for ensuring the conditions in the step are met. • An Enter block with Gosu code to execute on entering the step. • Any number of Event objects that generate on entering the step. • Any number of Notification objects that generate on entering the step. • An Exit block with Gosu code to execute on leaving a step. <p>Several of these steps can contain other, step-specific, components:</p> <ul style="list-style-type: none"> • An ActivityStep can contain any number of Activity steps that generate on entering the step. • An AutoStep or ActivityStep can contain any number of GO branches which lead from this step to another step. • A ManualStep can contain any number of TRIGGER branches which lead from this step to another step, if something or someone from outside the workflow system manually invokes it. (This happens typically through the PolicyCenter interface.) • A ManualStep can contain any number of TIMEOUT branches that lead to another step after the elapse of a certain time.
Outcome	One or more	A specialized step that indicates a last step out of which no branch leaves.
<Finish>	Zero or one	Structural. An optional code block that contains Gosu code to perform any last cleanup after the workflow reaches an Outcome.

For more information on the **Workflows** editor, see “Using the Workflow Editor” on page 367.

Workflow Gosu

Workflow elements Start, Finish, Enter, Exit, GO, TRIGGER, and TIMEOUT can all contain embedded Gosu. The Workflow engine executes this Gosu code any time that it executes that element. The specific order of execution is:

- The Workflow engine runs Start before everything else
- The Workflow engine runs Enter on entering a step.
- The Workflow engine runs Exit upon leaving a step. It runs Exit before the branch leading to the next step. Thus, the actual execution logic from Step A to Step B is to Exit A, then do the Branch, then Enter B.
- The Workflow engine runs GO, TRIGGER, TIMEOUT elements as it encounters them upon following a branch.
- The Workflow engine runs Finish after it runs everything else.

Within the Gosu block, you can access the currently-executing workflow instance as `Workflow`. If you need to use local variables, declare them with `var` as usual in Gosu. However, if you need a value that persists from one step to another, create it as an extension field on `Workflow` and set its value from scripting. You can also create subflows in the Gosu blocks.

The current bundle for workflow actions is the bundle that the application uses to load the `Workflow` entity instance. The expression `Workflow.Bundle` returns the workflow bundle. See “Bundles and Database Transactions” on page 335 in the *Gosu Reference Guide*.

Workflow Versioning

After you create a workflow script and make it active, it can create hundreds or even thousands of working instances in the PolicyCenter application. As such, you do not want to modify the script as actual existing workflow instances can possibly be running against it. (This is similar to modifying a program while executing it. It can lead to very unpredictable results.)

However, you might choose to modify a script. Then, you would want all newly created instances of the workflow to use your new version of the script.

Guidewire stores each workflow script in a separate XML file. By convention, Guidewire names each file a variant of `xxxWF.#.xml`:

- `xxx` the workflow name (which is camel-cased `LikeThis`)
- `#` is the version number of the workflow process (starting from 1)

Every newly created (copied) workflow script has a different version number from its predecessor. (The higher the version number, the more recent the script.) Thus, a script file name of `ManualExecutionWF.2.xml` means workflow type `ManualExecution`, version 2. As PolicyCenter creates new instances of the workflow script, it uses the most recent script—the highest-numbered one—to run the workflow instance against.

It is possible to start a specific workflow with a specific version number. For details, see “Instantiating a Workflow” on page 395.

The Workflow engine enforces the following rules in regards to version numbers:

- If you create a new workflow instance for a given workflow subtype, thereafter, the Workflow engine uses the script with the highest version number. PolicyCenter saves this number on the workflow instance as the `ProcessVersion` field.
- From then on, any time that the Workflow instance wakes up to execute, the Workflow engine uses the script with the same typecode and version number of the instance only.
- It is forbidden to have two workflow scripts with the same subtype and version number. The server refuses to start if you try.
- If a workflow instance cannot find a script with the right subtype and version number, it fails with an error and drops immediately into the `Error` state. (This might happen, perhaps, if someone inadvertently deleted the file or the file did not load for some reason.)

When to Create a New Workflow Version

Guidewire recommends, as a general rule, that you create a new workflow version under most circumstances if you modify a workflow. For example:

- If you add a new step to the workflow, then create a new workflow version.
- If you remove an existing step from the workflow, then create a new workflow version.
- If you change the step type, for example, from Manual to an automatic step type, then create a new workflow version.

More specifically, for each workflow:

- PolicyCenter records the current step of an active workflow in the database. Each change to the basic structure of a workflow requires a new version.
- PolicyCenter records the branch that an active workflow selects in the database. A change to the Branch ID requires a new version.
- PolicyCenter records the activity associated with an Activity step in the database. A change to an Activity definition requires a new version.
- PolicyCenter records the trigger activity that occurs in an active workflow in the database. A removal of a trigger requires a new workflow version.
- PolicyCenter records the `messageID` of each workflow message in the database. A modification to a `MessageStep` requires a new workflow version.

You do not need to create a new workflow version if you modify a constant such as the timeout value in the `TIMEOUT` step. PolicyCenter does record the wake-up time (for a `TIMEOUT` step) that it calculates from the timeout time in the database. However, changing a timeout value does not affect workflows that are already on that step. Therefore, you do not need to create a new workflow version.

If you do modify a workflow, be aware that:

- If you convert a manual step to an automatic step, it can cause issues for an active workflow.
- If you reduce a timeout value, any active workflows that have already hit that step will only wait the previously calculated time.

IMPORTANT If there is an active workflow on a particular step, do not alter that step without versioning the workflow.

Workflow Localization

At the start of the workflow execution, the Workflow engine evaluates the workflow locale and uses that locale for notes, documents, templates, and similar items. However, it is possible to set a workflow locale that is different from the default application locale through the workflow editor. This change then affects all notes, documents, templates, email messages, and similar items that the various workflow steps create or use.

You can also:

- Set a different locale for any spawned subworkflows.
- Set a locale for a Gosu block that a workflow executes.
- Set Studio to display a workflow step name in a different locale.

See “Localizing Guidewire Workflow” on page 65 in the *Globalization Guide* for details.

To set a workflow locale

To view or modify the locale for a workflow, click in the background area of the layout view. This opens a properties area at the bottom of the screen. Enter a valid `ILocale` type in the `Locale` field to set the overall locale for a workflow. See “Localizing Guidewire Workflow” on page 65 in the *Globalization Guide* for details.

Workflow Structural Elements

A workflow (or, more technically, a workflow XML script) contains a number of elements that perform a structural function in the workflow. For example, the `<Start>` element designates which workflow step actually initiates the workflow. Studio indicates the structural blocks by surrounding the block name with brackets in the workflow outline. (This reflects the XML-basis for these blocks.)

The workflow structural blocks include the following:

- <Context>
- <Start>
- <Finish>

<Context>

Every workflow begins with a <Context> block. You use it to conveniently define symbols that apply to the workflow. You can use these symbols over and over in that workflow. For example, suppose that you extend the Workflow entity and add User as a foreign key. Then, you can define the symbol user for use in the workflow script with the value Workflow.User.

Within the workflow, you have access to additional symbols, basically whatever the workflow instance knows about. For example, you can define a symbol such that inserting Cancellation into the text workflow actually inserts Workflow.PolicyPeriod.Cancellation.

Defining Symbols

You must specify in the context any foreign key or parameter that the workflow subtype definition references. To access the <Context> element, select it in the outline view. You add new symbols in the property area at the bottom of the screen.

Field	Description
Name	The name to use in the workflow process for this entity.
Type	The Guidewire entity type.
Value	The instance of the entity being referenced.

<Start>

The <Start> structural block defines the step on which the workflow starts. To set the first step, select <Start> in the outline view (center pane). In the properties pane at the bottom of the screen, choose the starting step from the drop-down list of steps. Studio displays the downward point of a green arrow on the step that you chose.

This element can optionally contain Gosu code to set up the workflow or its business data.

<Finish>

The <Finish> structural block is an optional block that contains Gosu code to perform any last cleanup after the workflow reaches an **Outcome**.

Common Step Elements

It is possible for each step in the workflow to also contain some or all of the following:

- Enter and Exit Scripts
- Asserts
- Events
- Notifications
- Branch IDs

The PolicyCenter Administration tab displays the current step for each given workflow instance.

Enter and Exit Scripts

A workflow step can have any amount of Gosu code in the Enter and Exit blocks to define what to do within that step. (Enter Script Gosu code is far more common.) To access the enter and exit scripts block, select a workflow step and view the properties tab at the bottom of the screen.

Enter Script	Gosu code that the Workflow engine runs just after it evaluates any Asserts (conditions) on the step. (That is, if none of the asserts evaluate to false. If this happens, the Workflow engine does not run this step.)
Exit Script	Gosu code that the Workflow engine runs as the final action on leaving this step.

For example, you could enter the following Gosu code for the enter script:

```
var msg = "Workflow " + Workflow.DisplayName + " started at " + Workflow.enteredStep
print(msg)
```

Note: If you rename a property or method, or change a method signature, and a workflow references that property or method in a Gosu field, PolicyCenter throws ParseResultsException. This is the intended behavior. You must reload the workflow engine to correct the error (**Internal Tools** → **Reload** → **Reload Workflow Engine**).

Asserts

A step can have any number of Assert condition statements. An Assert executes just before the Enter block. If an Assert fails, the Workflow engine throws an exception and handles it like any other kind of runtime exception. To access the Assert tab, select a workflow step.

Condition	Each condition must evaluate to a Boolean value.
Error message	If a condition evaluates to false, then the Workflow engine logs the supplied error message.

For example, you could add the following assert condition and error message to log if the assertion fails:

Condition

```
Workflow.currentAction == "start"
```

Error message to log if assertion fails

```
"Some error message if condition is false"
```

Events

A step can have any number of Event elements associated with it. An Event runs right after the Enter block, and generates an event with the given name and the business object. To access the Events tab, select a workflow step.

Entity Name	Entity on which to generate the event. This must a valid symbol name. See “<Context>” on page 379 for a discussion on how to use entity symbols in workflow Gosu.
Event Name	Name of the event to generate. This must be a valid event name. <ul style="list-style-type: none"> • For general information on events, see “Messaging and Events” on page 289 in the <i>Integration Guide</i>. • For what constitutes a valid event name, specifically see “List of Messaging Events in PolicyCenter” on page 309 in the <i>Integration Guide</i>.

For example:

Entity Name	account
Event Name	someEvent

Notifications

A step can have any number of non-blocking **Notification** activities. A notification in workflow terms is an activity that PolicyCenter sends out, but which does not block the workflow from continuing. PolicyCenter only uses it to notify you of something. The Workflow engine generates any notifications immediately after it executes the **Enter** code, if any. See “**ActivityStep**” on page 383 for more information on activity generation.

Name	Name of the activity.
Pattern	Activity pattern code. This must be a valid activity pattern as defined through Guidewire PolicyCenter.
Init	Optional Gosu code that the Workflow engine executes immediately after it creates the activity. Typically, you use this code to assign the activity. If you do not explicitly assign the activity, the Workflow engine auto-assigns the activity.

For example:

Name	notification
Pattern	general_reminder

Branch IDs

A branch is a transition from one step to another. Every branch has an ID, which is its reference name. An ID is necessary because the Workflow instance sometimes needs to persist to the database which branch it is trying to execute. (This can happen, for example, if an error occurs in the branch and the workflow drops into the **Error** state). A branch ID must be unique within a given step.

Generally, as you enter information in a dialog to define a step, you also need to enter branch information as well.

Basic Workflow Steps

Guidewire uses the following steps (or blocks) to create a workflow:

- AutoStep
- MessageStep
- ActivityStep
- ManualStep
- Outcome

AutoStep

An **AutoStep** is a step that PolicyCenter guarantees to finish immediately. That is, it does not wait for anything else such as an activity, a manual trigger, or a timeout before continuing to the next step. The **Workflows** editor indicates an autostep with an arrow icon in the box the represents that step.



Each **AutoStep** step must have at least one **G0** branch. (It can have more than one, but it must have at least one.) Each **G0** branch that leaves an **AutoStep** step—except for the last one listed in the XML code—must contain a condition that evaluates to either Boolean **true** or **false**.

After the **AutoStep** completes its **Assert**, **Enter**, and **Activity** blocks, it goes through its list of **G0** branches (from top to bottom in the XML code):

- It picks the first **G0** branch for which the condition evaluates to **true**.

- It picks the last GO element (without a condition) if none of the other GO branches evaluate to `true`.

At that point, it executes the `Exit` block and proceeds to the step specified by the winning GO element.

To create a new auto step

1. Right-click in the workflow workspace, and select **New AutoStep**.

2. Enter the following fields:

Field	Description
Step ID	ID of the step to create.
ID	ID of a branch leaving this step. It defaults to the <code>To</code> value if you do not supply a value.
To	ID of the step to which the workflow goes if the condition specified for this branch evaluates to <code>true</code> .

For example:

Step ID	Step1
ID	-
To	DefaultOutcome

3. Click on your newly created step. It is possible that there are additional tabs to fill out in the properties area at the bottom of the screen. See “Common Step Elements” on page 379 for information on the various tabs.

MessageStep

A **MessageStep** is a special-purpose step designed to support messaging-based integrations. It automatically generates and sends a single integration message and then stops the workflow until the message completes. (Typically, this is through receipt of an ack return message.) After the message completes, the workflow resumes automatically.

The **Workflows** editor indicates an message step with a mail icon in the box the represents that step.



Just before running the `Enter` block, the Workflow engine creates a new message and assigns it to `Workflow.Message`. Use the `Enter` block to set the payload for the message. After the `Enter` block finishes, the workflow commits its bundle and stops. This commits the message. At this point, the messaging subsystem picks up the message and dispatches it.

If something acknowledges the message (either internal or external), PolicyCenter stores an optional response string (supplied with the ack) on the message in the `Response` field. PolicyCenter then does the following:

- It copies the message into the `MessageHistory` table
- It updates the workflow to null out the foreign key to the original message and establishes a foreign key to the new `MessageHistory` entity.

It then resumes the workflow (by creating a new work item).

There can be any number of GO branches that leave a message step (but only GO branches). As with AutoStep, the Workflow engine evaluates each GO condition, and chooses the first one that evaluates to `true`. If none evaluate to `true`, the Workflow engine takes the branch with no condition attached to it.

To create a new message step

1. Right-click in the workflow workspace, and select **New MessageStep**.

2. Enter the following fields:

Field	Description
Step ID	ID of the step to create.
Destination ID	ID of the destination for the message. This must be a valid message destination ID as defined through the Studio Messaging editor.
EventName	Event name on the message.
ID	ID of a branch leaving this step. It defaults to the To value if you do not supply a value.
To	ID of the step to which the workflow goes if the condition specified for this branch evaluates to true.

For example:

Step ID	Step4
Dest ID	89
Event Name	EventName
ID	
To	DefaultOutcome

3. Click on your newly created step. It is possible that there are additional tabs to fill out in the properties area at the bottom of the screen. See “Common Step Elements” on page 379 for information on the various tabs.

ActivityStep

An **ActivityStep** is similar an **AutoStep**, except that it can use any of the branch types—including a **TRIGGER** or a **TIMEOUT**—to move to the next step. However, before an **ActivityStep** branches to the next step, it waits for one or more activities to complete. PolicyCenter indicates the termination of an activity by marking it one of the following:

- Completed (which includes either being approved or rejected)
- Skipped
- Canceled

Activities are a convenient way to send messages and questions asynchronously to users who might not even be logged into the application.

The **Workflows** editor indicates an activity step with a person icon in the box the represents that step.



Within an **ActivityStep**, you specify one or more activities. The Workflow engine creates each defined activity as it enters the step. (This occurs immediately after the Workflow engine executes the **Enter Script** block, if there is one.) The activity is available on all steps.

The only difference between an **Activity** and a **Notification** within a workflow is that:

- An **Activity** pauses the workflow until all the activities in the step terminate.
- A **Notification** does not block the workflow from continuing.

If more than one **Activity** exists on an **ActivityStep**, then the Workflow engine generates all of them immediately after the **Enter** block (along with any events or notifications). The step then waits for all of the activities to terminate. If desired, an **ActivityStep** can also contain **TIMEOUT** and **TRIGGER** branches as well. In that case, if a timeout or a trigger on the step occurs, then the workflow does not wait for all the activities to complete before leaving the step.

After PolicyCenter marks all the activities as completed, skipped or canceled, the **ActivityStep** uses one or more GO branches to proceed to the next step. There can be any number of GO branches that leave an activity step. As with AutoStep, the Workflow engine evaluates each GO condition, and chooses the first one that evaluates to true. If none evaluate to true, the Workflow engine takes the branch with no condition attached to it.

Notice that it is possible for the condition statement of a GO branch to reference a generated Activity by its logical name. For instance, it is possible that you want to proceed to a different step depending on whether PolicyCenter marks the Activity as completed or canceled.

To create a new activity step

1. Right-click in the workflow workspace, and select **New ActivityStep**.
2. The dialog contains the following fields:

Field	Description
Step ID	The ID of the step to create.
Name	Name of the activity.
Pattern	Activity pattern code. This must be a valid activity pattern as defined through Guidewire PolicyCenter.
ID	ID of a branch leaving this step. It defaults to the To value if you do not supply a value.
To	ID of the step to which the workflow goes if the condition specified for this branch evaluates to true.

3. Click on your newly created step and open the **Activities** tab at the bottom of the screen. After you create the **ActivityStep**, you need to create one or more activities. (Each **ActivityStep** must contain at least one defined activity.) These fields on the **Activities** tab have the following meanings:

Name	Name of the activity.
Pattern	Activity pattern code value. This must be a valid activity pattern code as defined through Guidewire PolicyCenter. To view a list of valid activity pattern codes, view the ActivityPattern typelist. Only enter a value in the Pattern field that appears on this typelist. For example: <ul style="list-style-type: none"> • approval • approvaldenied • general • ...
Init	Gosu code that the Workflow engine executes immediately after it creates the activity. Typically, you use this code to assign the activity. If you do not explicitly assign the activity, the Workflow engine auto-assigns the activity. For example, the following initialization Gosu code creates an activity and assigns it SomeUser in SomeGroup. <pre>Workflow.initActivity(Activity) Activity.autoAssign(SomeGroup, SomeUser)</pre> <p>The initialization code creates an activity based on the activity pattern that you set in the Pattern field.</p>

ManualStep

A **ManualStep** is a step that waits for an external TRIGGER to be invoked or a TIMEOUT to pass. Unlike **AutoStep** or **ActivityStep**, a **ManualStep** must not have, and cannot have, GO branches leaving it. However, it can have zero or more TRIGGER branches or zero, or more, TIMEOUT branches. It must have at least one of these branches. Otherwise, there would be no way to leave this step.

The **Workflows** editor indicates a manual step with an hour-glass icon in the box the represents that step.



Manual Step with Timeout

If you specify a *timeout* for this step, then you also need to specify one of the following. (See also “TIMEOUT” on page 390 for more discussion on these two values.)

Time Delta	The amount of time to wait or pause before continuing. Enter an integer number with its units (3600s, for example).
Time Absolute	A fixed point in time, as defined by a Gosu expression that resolves to a date. You can use the Gosu code to define the date, as in the following: <code>PolicyPeriod.Cancellation.CancelProcessDate</code> Or, you can use Gosu to calculate the point in time, as in the following: <code>PolicyPeriod.PeriodStart.addDays(-105)</code>

This defines the terms of the TIMEOUT branch that leaves this step. To view these details later, click the branch (the link) between the two steps.

Manual Step with Trigger

If you specify a *trigger* for this step, then you need only enter the branch information. This defines the terms of the TRIGGER branch that leaves this step. To view these details later, click the branch (the link) between the two steps.

To create a new manual step

1. Right-click in the workflow workspace, and select **New ManualStep**.
2. Enter the following fields. What you see in the dialog changes slightly depending on the value you set for **Type** (TIMEOUT or TRIGGER).

Field	Description
Step ID	ID of the step to create.
Type	Name of the activity.
ID	If you select the following Type value: <ul style="list-style-type: none">Trigger: A valid trigger key as defined in typelist <code>WorkflowTriggerKey</code>.Timeout: ID of a branch leaving this step. It defaults to the To value if you do not supply a value.
To	ID of the step to which the workflow goes if the condition specified for this branch evaluates to true.
Time Delta	Specifies a fixed amount of time to pause before continuing. For example, the following sets the wait time to 60 minutes (one hour): 3600s,
Time Absolute	Specifies a fixed point in time. For example, the following sets the point to continue to after the policy <code>CancelProcessDate</code> : <code>PolicyPeriod.Cancellation.CancelProcessDate</code>

If the `WorkflowTriggerKey` typelist does not contain any trigger keys, then you do not see the Trigger option in the dialog.

3. Click on your newly created step. It is possible that there are additional tabs to fill out in the properties area at the bottom of the screen.

Outcome

An **Outcome** is a special step that has no branches leading out of it. It is thus a final or terminal step. If a workflow enters any **Outcome** step, it is complete. It is possible (and likely) for a workflow to have multiple outcomes or final steps.

The **Workflows** editor indicates an outcome step with a gray bar in the box to indicate that this is a final step.



After the Workflow engine successfully enters an **Outcome** step (meaning that the Workflow engine successfully executes the **Enter** block of the **Outcome** step), it does the following:

1. The workflow generates all the listed events and notifications.
2. It executes the **<Finish>** block of the workflow process.
3. It changes the state of the workflow instance to **Completed**.

You must structure each workflow script so that its execution eventually and inevitably leads to an **Outcome**. Otherwise, you risk infinitely-running workflows, which means that the load on the Workflow engine can increase linearly over time, crippling performance.

To create a new outcome step

1. Right-click in the workflow workspace, and select **New Outcome**.
2. Enter a step ID in the **New Outcome** dialog.
3. Click on your newly created step. It is possible that there are additional tabs to fill out in the properties area at the bottom of the screen.

Step Branches

A branch is a transition from one step to another. There are multiple kinds of elements that facilitate branching to another step. They are:

- GO
- TRIGGER
- TIMEOUT

The **Workflows** editor indicates a branch by linking two steps with a line and placing one of the following icons on the line to indicate the branch type.

Type	Icon	Description
GO		A branch or transition to another workflow step. It occurs only within an AutoStep workflow step. <ul style="list-style-type: none"> • If there is only a single GO branch within the workflow step, branching occurs immediately upon workflow reaching that point. • If there are multiple GO branches within the workflow step, all GO branches (except one) must contain conditional logic. The workflow then determines the appropriate next step based on the defined conditions.
TRIGGER		A branch or transition to another workflow element. It occurs only within a ManualStep workflow step. Branching occurs only upon manual invocation from outside the workflow.
TIMEOUT		A branch or transition to another workflow element. It occurs only within a ManualStep workflow step. Branching to another workflow step occurs only after the passing of a specific time interval.

All branch elements contain a **To** value that indicates the step to which this branch leads. It can also contain an optional embedded Gosu block for the Workflow engine to execute if a workflow instance follows that branch.

How a workflow decides which branch to take depends entirely on the type of the branch. However, the order is always the same:

- The Workflow engine executes the **Enter** block for a given step and generates any events, notifications, and activities (waiting for these activities to complete).

- The Workflow engine attempts to find the first branch that is ready to be taken. It starts with the first branch listed for that step in the outline view, then moves to evaluate the next branch if the previous branch is not ready.
- If no branch is ready (which is possible only on a `ManualStep`), the workflow waits for one to become ready.
- After the Workflow engine selects a branch, it runs the `Exit` block, then executes the Gosu block of the branch.
- Finally, the workflow moves to the next step and begins to evaluate it.

Working with Branch IDs

Every branch also has an ID, which is its reference name. An ID is necessary because the Workflow instance sometimes needs to persist to the database which branch it is trying to execute. (This can happen, for example, if an error occurs in the branch and the workflow drops into the `Error` state). A branch ID must be unique within a given step.

If you do not specify an ID for a branch (which occurs frequently), the workflow uses the value of `nextStep` attribute as a default. This works well except in the special case in which you have more than one branch leading from the same `Step A` to the same `Step B`. (This can happen, for example, if you want to OR multiple conditions together, or if you want different Gosu in the different branches but the same `nextStep`.) In that case, you must add an ID to each of those branches. Studio complains with a verification error upon loading (or reloading) the workflow scripts if you do not do this.

Do the following to assign an ID to each type of branch:

Type	Action to take
GO	Optionally add an ID to a GO branch. If you do not provide one, Studio defaults the ID to the value of the <code>nextStep</code> attribute. However, Guidewire recommends that you create specific IDs if there are multiple GO branches that all move to the same next step.
TRIGGER	Always add an ID to a TRIGGER branch. Guidewire requires this as you must invoke a trigger explicitly. You must use a value from the <code>WorkflowTriggerKey</code> typelist for the branch ID.
TIMEOUT	Optionally add an ID to a TIMEOUT branch. If you do not provide one, Studio defaults the ID to the value of the <code>nextStep</code> attribute.

GO

The simplest kind of branch is `GO`. It appears on `AutoStep`, `ActivityStep` and `MessageStep`. There can be a single `GO` branch or a list of multiple `GO` branches. If there is a single `GO` branch, then you need only specify the `To` field and any optional Gosu code. The Workflow engine takes this `GO` branch immediately as it checks its branches.

The `Workflows` editor indicates a `GO` branch with an arrow icon superimposed on the line that links the two steps. (That is, the initial `From` step and the `To` step to which the workflow goes if the `GO` condition evaluates to `true`.)

To access the dialog that defines the `GO` branch, right-click the starting step—in this case, `CheckOnOrder`—and select `New GO` from the menu. (Studio only displays those choices that are appropriate for that step.) This dialog contains the following fields:

Field	Description
Branch ID	ID of the branch to create
From	ID of the step on which the <code>GO</code> branch starts.
To	ID of the step on which the <code>GO</code> branch starts.

As discussed (in “Working with Branch IDs” on page 387), it is not necessary to enter a branch ID. However, if you create multiple GO branches from a step, then you must enter a unique ID for each branch.

After you create the GO branch, click on the link (line) that runs between the two steps. You see a dialog that contains the following fields:

Field	Description
Branch ID	Automatically generated.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.
Arrow Visible	Show an arrow head on the branch line to indicate direction.
Description	Description of this branch.
Condition	Must evaluate to either true or false.
Execution	Gosu code to execute if the Workflow engine takes this branch.

Notice that this branch definition sets a condition. The **From** and **To** fields set the end-points for the branch.

If there are multiple GO branches, all the GO branches except one must define a condition that evaluates to either Boolean true or false. The Workflow engine decides which GO branch to take by evaluating the GO branches from top to bottom (within the XML step definition). It selects the first one whose condition evaluates to true. If none of the conditions evaluate to true, then the Workflow engine uses the GO branch that does not have a condition. A list of GO branches is thus like a switch programming block or a series of if...else... statements, with the default case at the bottom of the list.

Infinite Loops

Beware of infinite, immediately-executing cycles in your workflow scripts. For example:

From	To
StepA	StepB
StepB	StepA

If the steps revolve in an infinite loop, the Workflow engine only catches this after 500 steps. This can cause other problems to occur.

TRIGGER

Another kind of branch is TRIGGER, which can appear in a `ManualStep` or an `ActivityStep`. It also has a **To** field and an optional embedded Gosu block. However, instead of a condition checking to see if a certain Gosu attribute is true, someone or something must manually invoke a TRIGGER from outside the workflow infrastructure. (Typically, this happens from either PolicyCenter interface or from a Gosu call.) Guidewire requires a branch ID field on all TRIGGER elements, as outside code uses the ID to manually reference the branch.

Unlike all other the IDs used in workflows, TRIGGER IDs are not plain strings but typelist values from the extendable `WorkflowTriggerKey` typelist. This provides necessary type safety, as scripting invokes triggers by ID. However, it also means that you must add new typecodes to the typelist if you create new trigger IDs.

Invoking a Trigger

How does one actually invoke a TRIGGER? Almost anything can do so, from Gosu rules and classes to the PolicyCenter interface. Typically, in PolicyCenter, you invoke a trigger through the action of toolbar buttons in a wizard. This is done through a call to the `invokeTrigger` method on `Workflow` instances. (As it is also a scriptable method, you can call it from Gosu rules and the application PCF pages.) See “The `invokeTrigger` Method” on page 398 for a discussion of the `invokeTrigger` method and its parameters.

Internally, the method works by updating the (read-only) database field `triggerInvoked` on `Workflow` to save the ID. (See the PolicyCenter *Data Dictionary* entry on `Workflow`.)

The Workflow engine then *wakes up* the workflow instance and the TRIGGER inspects the `triggerInvoked` field to see if something invoked the trigger. Depending on how you set the `invokeTrigger` method parameters, the Workflow engine handles the result of the TRIGGER either synchronously or asynchronously.

Creating a Trigger Branch

To access the TRIGGER branch dialog, right-click the starting step and select **New Trigger** from the menu. (Studio only displays those choices that are appropriate for that step.) This dialog contains the following fields:

Field	Description
Branch ID	Name of this branch as defined in the <code>WorkflowTriggerKey</code> type list. Select from the drop-down list.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.

After you create the branch, click on the link (line) that runs between the two steps. You see the following fields, which are identical to those used to define a GO branch:

Field	Description
Branch ID	Automatically generated.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.
Arrow Visible	Show an arrow head on the branch line to indicate direction.
Description	Description of this branch.
Condition	Must evaluate to either true or false.
Execution	Gosu code to execute if the Workflow engine takes this branch.

Trigger Availability

Simply because you define a TRIGGER on a `ManualStep` does not mean it is necessarily available. You can restrict trigger availability in the following different ways:

- You can specify user access permission through the use of the `Permission` field.
- You can add any number of `Available` conditions on the `Available` tab to further restrict availability. If the condition expression evaluates to `true`, the trigger is available. Otherwise, it is unavailable.

For example (from PolicyCenter), the following Gosu code indicates that the workflow can only take this branch if a user has permission to rescind a policy. (The condition evaluates to `true`.)

```
PolicyPeriod.CancellationProcess.canRescind().Okay
```

TIMEOUT

Another kind of branch is TIMEOUT, which (like TRIGGER) can appear on ManualStep or an ActivityStep. You still have a To field and optional Gosu block. However, instead of using a condition to determine how to move forward, the Workflow engine executes the TIMEOUT element after the elapse of a specified amount of time.

You can use a TIMEOUT in the following ways:

- As the default behavior for a stalled workflow. For example:

Do x if PolicyCenter has not invoked a trigger for a certain amount of time.

- As a deliberate delay. For example:

Go to sleep for 35 days.

You can specify the time to wait using one of the following attributes. (Studio complains if you use neither or both.)

- timeDelta
- timeAbsolute.

The Time Delta Value

The Time Delta value specifies an amount of time to wait, starting from the time the Workflow instance successfully enters the step. (The wait time starts immediately after the Workflow engine executes the Enter Script block for the step.) You specific the time to wait with a number and a unit, for example:

- 100s for 100 seconds
- 15m for 15 minutes
- 35d for 35 day

You can also combine numbers and units, for example, 2d12h30m for 2 days, 12 hours, and 30 minutes.

The Time Absolute Value

Often, you do not want to wait a certain amount of time. Instead, you want the step to time out after passing a certain point relative to a date in the business model (for example, five days after a specific event occurs). In that case you can set the Time Absolute value, which is a Gosu expression that must resolve to a date.

IMPORTANT Do not use the current time in a Time Absolute expression. The Workflow engine re-evaluates this expression each time it checks TIMEOUT. For example, the time-out never ends for the following expression, `java.util.Date.CurrentDate + 1`, as the expression always evaluates to the future.

Creating a Timeout Branch

The following graphic illustrate how you define a Timeout branch in the Workflows editor. To access the Timeout branch dialog, right-click the starting step and select New Timeout from the menu. Notice that you must enter either time absolute expression or a time delta value. This dialog contains the following fields:

Field	Description
Branch ID	Name you choose for this branch.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.
Time Delta	Time to wait, starting from the time the Workflow instance successfully enters the step.
Time Absolute	Gosu expression that must resolve to a fixed date.

After you create the branch, click on the link that runs between the two steps. You see the following fields:

Field	Description
Branch ID	Automatically generated.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.
Arrow Visible	Show an arrow head on the branch line to indicate direction.
Time Delta	Time to wait, starting from the time the Workflow instance successfully enters the step.
Time Absolute	Gosu expression that must resolve to a fixed date.
Execution	Gosu code to execute if the Workflow engine takes this branch.

Creating New Workflows

To create a new workflow, you can do the following:

Action	Description
Cloning an Existing Workflow	Creates an exact copy of an existing workflow type, with the same name but with an incremented version number. (This process clones the workflow with highest version number, if there multiple versions already exist.) Perform this procedure if you merely want a new version of an existing workflow.
Extending an Existing Workflow	Creates a new (blank) workflow with a name of your choice based on the workflow type of your choice.

Cloning an Existing Workflow

Cloning an existing workflow is a relatively simple process. Also, if you clone an existing, fully built workflow, then you can leverage the work of the original workflow. However, you can only clone existing workflow types. You cannot use this method to create a new workflow type.

To clone an existing workflow

1. Open the **Workflows** node in the Project window tree.
2. Select an existing workflow type, right-click and select **New → Workflow** from the menu.

Studio creates a cloned, editable copy of the workflow process and inserts it under the workflow node with an incremented version number. You can then modify this version of the workflow process to meet your business needs.

Extending an Existing Workflow

To extend an existing workflow, you must create an **.eti** (extension) file and populate it correctly. To assist you, Studio provides a dialog in which you can enter the basic workflow information. You must then enter this information in the **.eti** file.

To extend an existing workflow

1. First, determine the workflow type that you want to extend.
2. Select **Workflows** in the Project window, right-click and select **Create metadata for a new workflow subtype** from the menu.

3. In the **New Workflow subtype metadata** dialog, enter the following:

Field	Description
Entity	The workflow object to create.
Supertype	The type or workflow to extend. You can always extend the Workflow type, from which all subtypes extend.
Description	Optional description of the workflow.
Foreign keys	Click the Add button to enter any foreign keys that apply to this workflow object.

4. Click **Gen to clipboard**. This action generates the workflow metadata information in the correct format and stores on the clipboard.
5. Expand the **Extensions** folder in the **Project** window.
6. Right-click the **Entity** folder and select **New → Entity** from the menu.
7. Enter the name of the file to create in the **New File** dialog. Enter the same value that you entered in the **New Workflow subtype metadata** dialog for **Entity** and add the **.eti** extension. Studio then creates a new **<entity>.eti** file. Open this file, right-click, and choose **Paste** from the menu. Studio pastes in the metadata workflow that you created in a previous step. For example, if you extend **Workflow** and create a new workflow named **NewWorkflow**, then you must create a new **NewWorkflow.eti** file that contains the following:
- ```
<?xml version="1.0"?>
<subtype desc="" entity="NewWorkflow" supertype="Workflow"/>
```
8. (Optional) To provide the ability to localize the new workflow, add the following line of code to this file (as part of the **subtype** element):
- ```
<typekey desc="Language" name="Language" typelist="LanguageType"/>
```
- Continuing the previous example, you now see the following:
- ```
<?xml version="1.0"?>
<subtype desc="" entity="NewWorkflow" supertype="Workflow">
 <typekey desc="Language" name="Language" typelist="LanguageType"/>
</subtype>
```
9. Stop and restart Guidewire Studio so that it picks up your changes.
- You now see **NewWorkflow** listed in the **Workflow** typelist.
  - You now see an **NewWorkflow** node under **Resources → Workflows**.
10. Select the **NewWorkflow** node under **Workflows**, right-click and select **New Workflow Process** from the menu. Studio opens an empty workflow process that you can modify to meet your business needs.

## Extending a Workflow: A Simple Example

This simple examples illustrates the following steps:

- Step 1: Extend an Existing Workflow Object
- Step 2: Create a New Workflow Process
- Step 3: Populate Your Workflow with Steps and Branches

### Step 1: Extend an Existing Workflow Object

To extend an existing workflow object, review the steps outlined in “Extending an Existing Workflow” on page 391. For this example, you create a new **ExampleWorkflow** object by extending (subtyping) the base **Workflow** entity.

**To extend a workflow object**

1. Create a new ExampleWorkflow.eti file and enter the following:

```
<?xml version="1.0"?>
<subtype desc="" entity="ExampleWorkflow" supertype="Workflow">
 <typekey desc="Language" name="Language" typelist="LanguageType"/>
</subtype>
```

2. Close and restart Studio.

You now see an ExampleWorkflow entry added to the Workflow typelist and a new ExampleWorkflow workflow type added to Workflows in the Resources tree.

**Step 2: Create a New Workflow Process**

Next, you need to create a new workflow process from your new ExampleWorkflow type.

**To create a new workflow process**

1. Select ExampleWorkflow from Workflows in the Project window.
2. Right-click and select New Workflow from the menu.

Studio opens an outline view and layout view for the new workflow process:

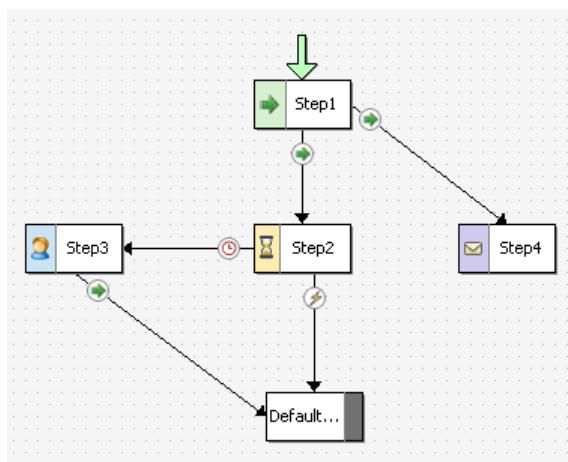
- The outline view contains the few required workflow elements.
- The layout view contains a default outcome (DefaultOutcome).

**Step 3: Populate Your Workflow with Steps and Branches**

Finally, to be useful, you need to add outcomes, steps, and branches to your workflow. This examples creates the following:

- A Step1 (AutoStep) with a default GO branch to the DefaultOutcome step, which you designate as the first step in the <Start> element
- A Step2 (ManualStep) with a TRIGGER branch to the DefaultOutcome step
- A Step3 (ActivityStep) with a GO branch to the DefaultOutcome step
- A TIMEOUT branch from Step2 to Step3, with a 5d time delta set
- A Step4 (MessageStep) with a GO branch from Step1 to Step4

The example workflow looks similar to the following:



This example does not actually perform any function. It simply illustrates how to work with the dialogs of the Workflows editor.

### To add steps and branches to a workflow

1. Right-click within an empty area in the layout view and select **New AutoStep** from the menu:
  - For **Step ID**, enter Step1.
  - Do not enter anything for the other fields.Studio adds your autostep to the layout view and connects Step1 to DefaultOutcome with a default GO branch.
2. Select <Start> in the outline view (middle pane):
  - Open the **First Step** drop-down in the property area at the bottom of the screen.
  - Select Step1 from the list. This sets the initial workflow step to Step1.
  - Save your work.
3. Right-click within an empty area in the layout view and select **New ManualStep** from the menu:
  - For **Step ID**, enter Step2.
  - For branch **Type**, select TRIGGER.
  - For trigger **ID**, select Cancel.The ID value sets a valid trigger key as defined in typelist WorkflowTriggerKey. If Cancel does not exist, then choose another trigger key. If no trigger keys exist in WorkflowTriggerKey, then you must create one before you can select TRIGGER as the type.
4. Select the GO branch (the line) leaving Step1:
  - In the property area at the bottom of the screen, change the **To** field from DefaultOutcome to Step2. Studio moves the branch to link the specified steps.
  - Realign the steps for more symmetry, if you choose.
5. Right-click within an empty area in the layout view and select **New ActivityStep** from the menu:
  - For **Step ID**, enter Step3.
  - For **Name**, enter ActivityPatternName.
  - For **Pattern**, enter NewActivityPattern.
6. Select Step3, right-click, and select **New TIMEOUT** from the menu:
  - For **Branch ID**, enter TimeoutBranch.
  - For **Time Delta**, enter 5d. This sets the absolute time to wait to five days.
  - For **To**, select Step3.Studio adds a branch from Step2 to Step3 and adds the timeout symbol to it.
7. Right-click within an empty area in the layout view and select **New MessageStep** from the menu:
  - For **Step ID**, enter Step4.
  - For **Dest ID**, enter 89 (or any valid message destination ID).
  - For **Event Name**, enter EventName.Studio adds the step to the layout view and creates a link between Step4 and DefaultOutcome.
8. Select the new link from Step4 to DefaultOutcome.
  - In the property area at the bottom of the screen, change **Arrow Visible** to **false** to delete this link.Studio removes the link (branch).
9. Select Step1, right-click, and select **New GO** from the menu:
  - For **Branch ID**, enter Step4.
  - For **To**, select Step4.Studio adds the new GO branch between Step1 and Step4.

## Instantiating a Workflow

It is not sufficient to create a workflow. Generally, you want to do something moderately useful with it. To perform work, you must instantiate your workflow and call it somehow.

Suppose, for example, that you create a new workflow and call it, for lack of a better name, `HelloWorld1`. You can then instantiate your workflow using the following Gosu:

```
var workflow = new HelloWorld1()
workflow.start()
```

### Starting a Workflow

There are multiple workflow `start` methods. The following list describes them.

<code>start()</code>	Starts the workflow.
<code>start(version)</code>	Starts the workflow with the specified process version.
<code>startAsynchronously()</code>	Starts the workflow asynchronously.
<code>startAsynchronously(version)</code>	Starts the workflow with the specified process version asynchronously.

For information on versioning works with workflow, see “Workflow Versioning” on page 377.

### Logging Workflow Actions

There are several different Gosu statements that you can use to view workflow-related information.

<code>gw.api.util.Logger.logInfo</code>	Statement written to the application server log
<code>Workflow.log</code>	Statements viewable in the PolicyCenter Workflow console

### See Also

- See Workflow Debugging, Logging, and Testing for more information.

## A Simple Example of Instantiation

The following example creates a trivial workflow named `HelloWorld1`. The objective of this example is not to show the branching structure that you can create in workflow. Rather, the purpose of this exercise is to construct the workflow, trigger the workflow, and examine the workflow in the PolicyCenter `Workflow` console. The example keeps the workflow as simple as possible. The workflow consists of the following components:

- `<Context>`
- `<Start>`
- `Step1`
- `Step2`
- `DefaultOutcome`
- `<Finish>`

### A Simple ClaimCenter Example

**Note:** This example uses business entities and rules that apply specifically to the Guidewire ClaimCenter application. However, the particular business objects are not important. What is more important is how you create and instantiate a workflow process.

For the workflow to run and do some work and appear on the workflow console, the example instantiates it from a Claim Update rule. If you attempt to instantiate the workflow from a link or button on a Claim view screen (`Claim Summary`, for example) the workflow executes but does not update anything. Also, it does not appear in the `Workflow` console.

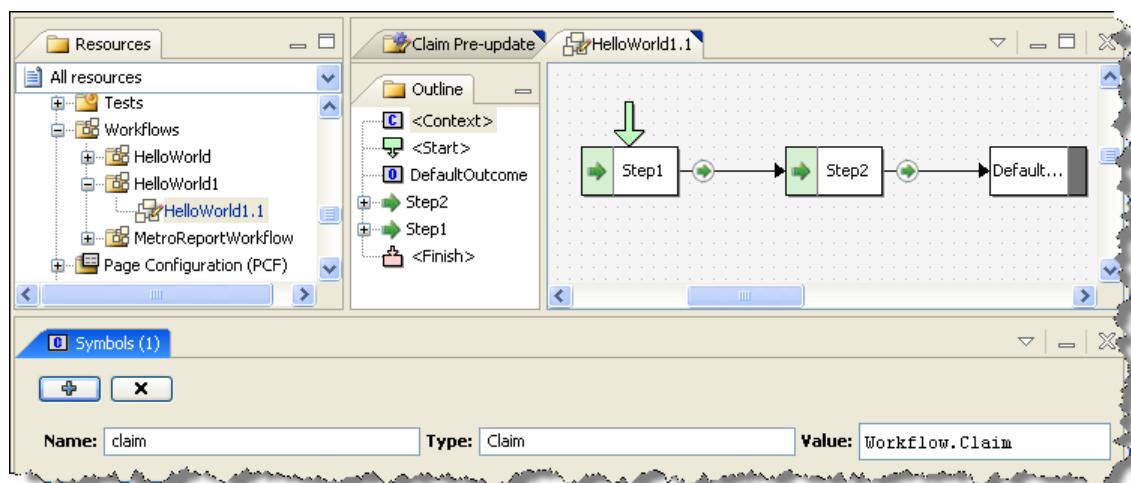
To cause updates to happen, the example instantiates the workflow from an **Edit** screen in ClaimCenter. It then calls a Claim Pre-Update Rule in Studio.

### To create a simple workflow and instantiate it

1. Create a `HelloWorld1.eti` file (in Extensions → Entity) and populate it with the following:

```
<?xml version="1.0"?>
<subtype desc="HelloWorld 1 Example Workflow"
 entity="HelloWorld1"
 supertype="ClaimWorkflow">
 <typekey desc="Language" name="Language" typelist="LanguageType"/>
</subtype>
```

2. Stop and restart Studio.
3. Select your new workflow type from the **Workflows** node. Right-click and select **New → Workflow Process**.
4. Create a simple workflow process similar to the following. It does not need to be complex, as it simply illustrates how to start a workflow from the ClaimCenter interface.



Notice that it has a `claim` symbol set in `<Context>`.

5. For Step1, add the following to the Enter block for that step:

```
gw.api.util.Logger.logInfo("HelloWorld1 step 1, step called ClaimNumber " + claim.ClaimNumber)
Workflow.log("HelloWorld Step 1", "HelloWorld1 step 1 entered: Claim Number " + claim.ClaimNumber)
```

6. For Step2, add the following to the Enter block for that step:

```
gw.api.util.Logger.logInfo("HelloWorld1 step 2, step called ClaimNumber " + claim.ClaimNumber)
Workflow.log("HelloWorld Step 2", "HelloWorld1 step 2 entered: Claim Number " + claim.ClaimNumber)
```

7. Create a simple Claim Pre-Update rule similar to the following:

- The *rule condition* specifies that the Workflow engine instantiates the workflow only if the claim `PermissionRequired` property is set to `fraudriskclaim`.
- The *rule action* instantiates the `HelloWorld1` workflow. It first tests for an existing `HelloWorld1` workflow that is not in the completed state and that has the same claim number as the one being updated. If it does not find a matching workflow, then the Workflow engine instantiates `HelloWorld1` and logs the information.

#### Rule Conditions:

```
claim.PermissionRequired=="fraudriskclaim"
```

#### Rule Actions:

```
gw.api.util.Logger.logInfo("Entering Pre-Update")

var hw_wf = claim.Workflows.firstWhere(\ c -> c.Subtype == "HelloWorld1"
 && (c as entity.HelloWorld1).State != "completed"
 && (c as entity.HelloWorld1).Claim.ClaimNumber==claim.ClaimNumber)
```

```
if (hw_wf == null) {
 gw.api.util.Logger.logInfo("# Studio instantiating HelloWorld1 and starting it!")
 var workflow = new entity.HelloWorld1()
 workflow.Claim = claim
 workflow.start()
}
```

8. Log into ClaimCenter and open any sample claim.
9. Navigate to the **Claim Summary** page, then select the **Claim Status** tab.
10. Click **Edit** and set the **Special Claim Permission** value to **Fraud risk**.
11. Click **Update**. This action triggers the `HelloWorld1` workflow.

#### To view the server console

1. Navigate to the application server console.
2. View the logger statements.

#### To view the Workflow console

1. Log into ClaimCenter using an administrative account.
2. Navigate to the **Administration** tab and select **Workflows** from the left-side menu.
3. Click **Search** in the **Find Workflows** screen. You do not need to enter any search information. Studio displays a list of workflows, including `HelloWorld1`.
4. Select `HelloWorld1` from the list and view its details.

## The Workflow Engine

The Workflow engine is responsible for processing a workflow. It does this by looking up and executing the appropriate Workflow Process Script. This script (often just called Workflow Process or Workflow Script) is an XML file that the Studio Workflow editor generates, and which you manage in Studio. The base configuration workflow scripts live in the `modules/config/workflow` directory.

### Distributed Execution

PolicyCenter uses a work queue to handle workflow execution. This, in simple terms, means that you can have a whole cluster of machines that:

- Wake up internal `Workflow` instances,
- Advance them as far as they can go,
- Then, let them go back to sleep if they need to wait on a timeout or activity.

Asynchronous workflow execution always works the same way:

1. PolicyCenter creates a `WorkflowWorkItem` instance to advance the workflow.
2. The worker instance picks up the work item.
3. The work item retrieves the workflow and advances it as far as possible (to a `ManualStep` or `Outcome`).

You can create a work item in any of the following different ways:

- By a call to the `AbstractWorkflow.startAsynchronously` method
- By invoking a trigger with `asynchronous = true`
- By completing a workflow-linked activity
- By the `Workflow` batch process, which queries for active workflows waiting on an expired timeout

- By a call to `AbstractWorkflow.resume`, typically initiated by an administrator using the workflow management tool

After the workflow advances as far as it can, PolicyCenter deletes the work item and execution stops until there is another work item.

## Synchronicity, Transactions, and Errors

To understand how error handling works in the internal Workflow engine, you must know whether the workflow is running synchronously or asynchronously.

### Synchronous and Asynchronous Workflow

It is possible to start workflow either synchronously or asynchronously. To do so, use one of the `start` methods described in “Instantiating a Workflow” on page 395. To review, these are:

- `start()`
- `start(version)`
- `startAsynchronously()`
- `startAsynchronously(version)`

If a workflow runs synchronously, then it continues to go through one `AutoStep` or `ManualStep` after another until it arrives at a stop condition. This advance through the workflow can encompass one or multiple steps. The workflow executes the current step (unless there is an error), and then continues to the next step, if possible. There can be many different reasons that a workflow cannot continue to the next step. For example:

- It can encounter an activity step (`ActivityStep`). This can result in the creation of one or more activities, causing the workflow to pause until the closure of all the activities.
- It can encounter a communication step (`MessageStep`). This can result in a message being sent to another system, causing the workflow to wait until receiving a response.
- It can encounter a step that stipulates a timeout (`ManualStep`). This causes the workflow to wait for the timeout to complete.
- It can encounter a step that requires a trigger (`ManualStep`). This causes the workflow to wait until someone (or something) activates the trigger.
- And, of course, ultimately, the workflow can run until it reaches an `Outcome`, at which point, it is done.

After pausing, the workflow waits for one of the following to occur:

- If waiting on one or more activities to complete, it continues after the closure of the last activity.
- If waiting for an acknowledgement of a message, it continues after receiving the appropriate response.
- If waiting on a timeout, it continues after the timeout elapses.
- If waiting on an external trigger, then someone or something must manually invoke a `TRIGGER` from outside the workflow infrastructure. This can happen either from the PolicyCenter interface (a user clicking a button) or from Gosu. In either case, this is done through a call to the `invokeTrigger` method on a `Workflow` instance.

The action of completing an activity or the receipt of a message response automatically creates a work item to advance the workflow. A background batch process checks for timeout elements. It is responsible for finding timed-out workflows that are ready to advance and creating a work item to advance them.

### The `invokeTrigger` Method

If a user (or Gosu code) invokes an available trigger (`TRIGGER`) on a `ManualStep`, the workflow can execute either synchronously or asynchronously. A Boolean parameter in the `invokeTrigger` method determines the execution type. This method takes the following signature:

```
void invokeTrigger(WorkflowTrigger triggerKey, boolean synchronous)
```

For example (from PolicyCenter):

```
policyPeriod.ActiveWorkflow.invokeTrigger(trigger, false)
```

The `trigger` parameter defines the TRIGGER to use. This must be a valid trigger defined in the `WorkflowTriggerKey` type list.

The `synchronous` value in this method has the following meanings:

<code>true</code>	(Default) Instructs the workflow to immediately execute in the current transaction and to block the calling code until the workflow encounters a new stopping point.
<code>false</code>	Instructs the workflow to run in the background, with the calling code continuing to execute. The workflow continues until it encounters a new stopping point.

### Trigger Availability

For a trigger to be available, the workflow execution sequence must select a branch for which both of the following conditions are true:

- A trigger must exist on the step.
- There is no other determinable path (which usually means that no timeout has already expired).

Thus, if both of these conditions are true, after an invocation to the `invokeTrigger` method, the Workflow engine starts to advance the workflow from the selected branch again.

### Invoking a Trigger

Invoking a trigger (either synchronously or asynchronously) does the following:

1. It updates the workflow. Any changes made to a transaction bundle that were committed by the actual invocation of the trigger, are committed.
2. It causes the workflow to create a log entry of the trigger request. If there is an error in the workflow advance, any request to the workflow to resume causes the process to start again. (See also “Workflow Administration” on page 401.)
3. If the Workflow engine determines that all the preconditions are met for continuing, it does the following:
  - a. It determines the *locale* in which to execute.  
This is the locale that PolicyCenter uses for display keys, dates, numbers, and other similar items. By default, this is the application default locale. It is important for the Workflow engine to determine the locale as it is possible to override this locale for any specific workflow subtype. You can also override the locale in the workflow definition on the workflow element. See “Localizing Guidewire Workflow” on page 65 in the *Globalization Guide* for more information.
  - b. It steps through each of the workflow steps (meaning that it performs all the actions within that step) until it cannot keep going.
  - c. It commits the transaction associated with the executed steps to the database.

### Error Handling and Transaction Rollback

If there is an error during a workflow step, the Workflow engine rolls the database back and leaves it in the state that it was. If working with an external system, you need to one of the following:

- You need to design the services in the external system, or,
- You need to use the Guidewire message subsystem to keep an external system state in synchronization with the application database state.

It is important to understand whether a workflow executes synchronously or asynchronously as it affects errors and transaction rollbacks:

Execution type	Application behavior
Synchronous	<p>If any exception occurs during <i>synchronous</i> execution, even after the workflow has gone through several steps, PolicyCenter rolls back all workflow steps (along with everything else in the bundle). The error cascades all the way up to the calling code (the code that started the workflow or invoked the trigger on the workflow).</p> <ul style="list-style-type: none"> <li>If you start the workflow or invoke the trigger from the PolicyCenter interface, PolicyCenter displays the exception in the interface.</li> <li>If some other code started the workflow, that code receives the exception.</li> </ul>
Asynchronous	<p>If any exception occurs during <i>asynchronous</i> execution (as it executes in the background), PolicyCenter logs the exception and rolls back the bundle, in a similar manner to the synchronous case.</p> <p>PolicyCenter then handles workflow retries in the standard way through the worker. PolicyCenter leaves the work item used to advance the workflow checked out. It simply waits until the <code>progressInterval</code> defined for the workflow work queue expires. At that point, a worker picks it up and retries it. The work queue configuration limits the number of retries. If all retries fail, PolicyCenter marks the work item as failed and it puts the workflow into the <code>Error</code> state. A workflow in the <code>Error</code> state merely sits idle until you restore it from the <b>Administration</b> tab within PolicyCenter. Restoring the workflow creates another work item.</p> <p>After you manually restore a workflow from an <code>Error</code> to an <code>Active</code> state, it again tries to resume whatever it was doing as it left off, typically:</p> <ul style="list-style-type: none"> <li>entering the step</li> <li>following the branch</li> <li>or, attempting to perform whatever it was doing at the time the exception occurred</li> </ul> <p>Of course, if you have not corrected the problem that caused the error, then the workflow can drop right back into <code>Error</code> state again. This is only after the work item performs its specified number of retries, however.</p>

## Guidelines

In practice, Guidewire recommends that you keep the following guidelines in mind as you work with workflows:

- If you invoke a workflow `TRIGGER`, do so synchronously if you need to make immediate use (in code) of the results of that trigger. For this reason, the PolicyCenter rendering framework typically always invokes the trigger synchronously. But notice that you only get immediate results from an `AutoStep` that might have executed. If the workflow encounters a `ManualStep` or an `ActivityStep`, it immediately goes into the background.
- If you complete an activity, it does not synchronously (meaning immediately) advance the workflow. Instead, a background process checks for workflows whose activities are complete and which are therefore ready to move forward. Guidewire provides this behavior, as otherwise, if an error occurs, the user who completes the activity sees the error, which is possibly confusing for that user.
- If you invoke a workflow `TRIGGER` from code that does not necessarily care whether there was a failure in the workflow, you need to invoke the `TRIGGER` asynchronously. (You do this by setting the `synchronous` value in the workflow method to `false`.) That way, the workflow advances in the background and any errors it encounters force the workflow into the `Error` state. The exception does not affect the caller code. However, the calling code creates an exception if it tries to invoke an unavailable or non-existent workflow `TRIGGER`. Messaging plugins, in particular, need to always invoke triggers asynchronously.

## Workflow Subflows

A workflow can easily create another child workflow in Gosu using the scriptable `createSubFlow` method on `Workflow`. There are multiple versions of this method:

```
Workflow createSubFlow(workflow)
Workflow createSubFlow(workflow, version)
```

A subflow has the same foreign keys to business data as the parent flow. It also has an edge foreign key reference to the caller `Workflow` instance, appropriately accessed as `Workflow.caller`. (If internal code, and not some other workflow, calls a *macro* workflow, this field is `null`.)

Each workflow also has a `subFlows` array that lists all the flows created by the workflow, including the completed ones. (This array is empty for workflows that have yet to create any subflows.) The Gosu to access this array is:

```
Workflow.SubFlows
```

You can use subflows to implement simple parallelism in internal workflows, which is otherwise impossible as a single workflow instance cannot be in two steps simultaneously. For example, it is possible for the macro flow to create a subflow in step A. It can then leave this subflow to do its own work, and only wait for it to complete in step E. It is your responsibility as the one configuring the macro workflow to decide how to react if a subflow drops into `Error` mode or becomes canceled for some reason.

#### See also

- “Creating a Locale-Specific Workflow SubFlow” on page 67 in the *Globalization Guide*

## Workflow Administration

You can administer workflow in any of the following ways:

- Through the PolicyCenter **Administration** → **Workflows** page
- Through the command line, for example, you can run a batch process to purge the workflow logs
- Through class `gw.webservice.workflow.IWorkflowAPI` (which the command line uses)

The most likely need for using the PolicyCenter **Administration** interface is error handling. Errors can be the following:

- A few workflows fail
- Or, in a worst case scenario, thousands fail simultaneously

Finding workflows that have not failed but have been idling for an extremely long time is also likely. A secondary use is just looking at all the current running flows to see how they work. Guidewire therefore organizes the **Administration** interface for workflow around a search screen for searching for workflow instances. You can filter the search screen, for example, by instance type, state (especially `Error` state), work item, last modified time, and similar criteria.

A user with administrative permissions can search for workflows from the **Administration** → **Workflows** page. However, to actually manage workflow, that user must have the `workflowmanage` permission. In the base PolicyCenter configuration, only the `superuser` role has this permission.

With the correct permission, you can do the following from the **Administration** → **Workflows** page:

- Search for a specific workflow or see a list of all workflows:
- Look at an individual workflow details, for example:
  - View its log and current step and action
  - View any open activities on the workflow
- Actively manage a workflow

### Manage Workflow

If you have the `workflowmanage` permission, PolicyCenter enables the following choices on the **Find Workflows** page:

- Manage selected workflows (active after you select one or more workflows)
- Manage all workflows (active at all times with the correct permission)

Choosing one of these options opens the **Manage Workflows** page. This page presents a choice of workflow and step appropriate commands that you can execute. It is only possible to select one command (radio button) at a time. Choosing either **Invoke Trigger** or **Timeout Branch** provides further selection choices.

Command	Description
Wait - max time (secs)	Select and enter a time to force the workflow to wait until either that amount of time has expired or the currently active work item is no longer active. (The work item has failed or has succeeded and has been deleted.)  This option is only available if there is a currently available work item on this workflow.
Invoke Trigger	Select to choose a workflow trigger to invoke. After selecting this command, PolicyCenter presents a list of available triggers from which to choose, if any are available on this workflow.
Suspend	Select to suspend any active workflows that are currently selected in the previous screen. After you execute this command, PolicyCenter suspends the selected workflows. This action is appropriate for all workflow and steps. However, PolicyCenter executes this command only against active workflows.
Resume	Select to resume workflow execution of any suspended workflows that are currently selected in the previous screen. This action is appropriate for all workflows and steps.
Timeout branch	Select to choose a workflow timeout branch. After selecting this command, PolicyCenter presents a list of timeout branches from which to choose, if any are available on this workflow.

After you make your selection and add any relevant parameters, clicking **Execute** immediately executes that command. Using these commands, you can:

- Restore workflows from the **Error** or **Suspended** state back to the **Active** state. However, if you have not corrected the underlying error, presumably a scripting error, the workflow might drop right back into **Error** mode.
- Force a waiting workflow to execute:
  - By setting the specific timeout branch
  - By setting a specific trigger
- Force an active workflow to wait for a specified amount of time

### Workflow Statistics Tab

PolicyCenter collects workflow statistics periodically and captures the elapse and execution time for individual workflow types and steps. You can search by workflow type and date range.

### Workflow and Server Tools

Those with access to the Server Tools, can also access the following:

Batch Process Info	Use to view information on the last run-time of a writer, and to see the schedule for its next run-time. From this page, you also have the ability to stop and start the scheduling of the writer.
Work Queue Info	Use to view information on a writer, what items it picked up and the workers. From this page, you also have the ability to notify, start and stop workers across the cluster.

## Workflow Debugging, Logging, and Testing

For more information on application logging, see “Configuring Logging” on page 23 in the *System Administration Guide*.

Debugging a workflow is a more challenging task than debugging the standard PolicyCenter interface flow, as most of the work happens asynchronously, away from any user. Currently, there is no way to set breakpoints in a workflow in a similar fashion to how you can set a breakpoint for a Gosu rule or class.

Guidewire does provide, however, workflow logging. Each instance of a workflow has its own internal log that you can view from within PolicyCenter. (You access this log from **Workflows** page by first by finding a workflow, then by clicking on the **Workflow Type** link.) This log includes successful transitions in the current step and action. It also contains any exceptions. Workflow can access this log, but PolicyCenter only commits these log message with the bundle.

Use the following logging method, for example, in an **Enter Script** block to log the current workflow step:

```
Workflow.log(summary, description)
```

The method returns the log entry (**WorkflowLogEntry**) that you can use for additional processing:

```
var workflowLog = Workflow.log("short description", "stack trace ...")
var summary = workflowLog.summary
```

### Process Logging

The following logging categories can be useful:

Category	Use for
WorkQueue	A category for general logging from the work queue.
WorkQueue.Instrumented	Capturing of runner state for a specific execution of the runner.
WorkQueue.Item	Logging (by workers) of each work item executed at the “info” level.
WorkQueue.Runner	Logging runners.

To write every message logged by every workflow, set the logging level of the workflow logger category to DEBUG (using **logging.properties**). The directive in the **logging.properties** file is:

```
log4j.category.Server.workflow=DEBUG
```

### Workflow Testing

It can often be difficult to test a workflow. This is especially true for one that is asynchronous and that requires the workflow to wait a specific amount of time before advancing to the next step. To facilitate testing, Guidewire PolicyCenter supports a testing clock that permits the advancing of time (for development-mode servers only). If you have permission, you can access this functionality from the (unsupported) PolicyCenter **Internal Tools** page. Depending on which clock you define in the **ITestingClock.xml** plugin registration file, you can do one of the following:

- Increment the current clock by a given period.
- Change the setting of the clock to a specific time. The clock remains at that time until another specific time is set.

### To enable the **ITestingClock** plugin

If the **ITestingClock** plugin is not already implemented, then you need to implement it.

1. In Studio, navigate to **Plugins** → **gw** → **plugin** → **system**.
2. Right-click **ITestingClock** and select **Implement**.
3. Click **Add** → **Java**.
4. Enter the following in the **Class** field.

```
com.guidewire.pl.plugin.system.internal.OffsetTestingClock
```



# Defining Activity Patterns

This topic discusses activity patterns, what they are, and how to configure them.

This topic includes:

- “What is an Activity Pattern?” on page 405
- “Pattern Types and Categories” on page 406
- “Using Activity Patterns in Gosu” on page 407
- “Calculating Activity Due Dates” on page 407
- “Configuring Activity Patterns” on page 408
- “Using Activity Patterns with Documents and Emails” on page 410
- “Localizing Activity Patterns” on page 411

## What is an Activity Pattern?

Activity patterns standardize the way that Guidewire PolicyCenter creates activities. Activity patterns describe the kinds of activities that people perform while handling policies within an organization. For example, reviewing and approving a policy renewal is a common activity. Thus, it has its own activity pattern that creates a reminder to perform this activity.

Patterns act as templates for creating activities. Activity patterns define the typical practices for each activity. For example, this is its name, its relative priority, and the standards for how quickly it is to complete (that is, its due dates). If a user (or a rule) adds an activity to the workplan for a policy, PolicyCenter uses the activity pattern as a template to set default values for the activity. (For example, an activity pattern can set the subject, priority, or target date for the activity.)

You can set up and customize the **Activity Patterns** that make sense for your policies business processes from the **Administration** tab in PolicyCenter. It is possible to create activities from activity patterns in different ways:

- You can manually create activities in PolicyCenter.
- A business rule or some other Gosu code create activities as part of generating workplans or while responding to escalations, policy exceptions, or other events.

- PolicyCenter automatically creates activities to handle manual assignment or approvals, for example.
- External applications create activities through API calls.

You can view the list of available **Activity Patterns** by selecting **New Activity** from the **Actions** menu on one of the **Summary** pages.

---

**IMPORTANT** After an activity pattern is in production, do not delete it as there can be old activities tied to it. Instead, edit the activity pattern and change the **Automated only** field to **Yes**. This prevents anyone from creating new activities of that type.

---

An activity pattern does not control how PolicyCenter assigns an activity. Instead, activity assignment methods in Gosu expressions control how PolicyCenter assigns an activity. Using the pattern name, the assignment methods determine to whom to assign the activity.

## Pattern Types and Categories

PolicyCenter applies a **type** attribute to every activity pattern. You can also use a **category** attribute to classify patterns into related groups. This topic describes how PolicyCenter makes use of these two attributes.

### Activity Pattern Types

Each activity pattern has a set type (for example, *General* or *Approval*). You can only add an activity pattern of type *General* through the PolicyCenter interface. An example of the use of a general activity pattern is an activity that generates a notification that reminds you to perform some task.

Guidewire defines a number of *internal* activity pattern types in the base configuration. All pattern types other than *General* are internal. Only internal PolicyCenter code can use an internal pattern type. Do not attempt to remove an internal activity pattern type as this can damage your installation. You can, however, customize attributes of the internal activity patterns, such as adjusting the due date.

#### The **ActivityType** Typelist

Guidewire defines activity pattern types in the **ActivityType** typelist. Guidewire defines this typelist as *final*. Typelists marked as final are internal typelists and used by internal application code. You cannot add typecodes to—or delete typecodes from—a typelist marked as final. You can, however, modify some of the fields on an existing typecode, if you wish. For more information on typelists marked as final, see “Internal Typelists” on page 268.

In the base configuration, Guidewire PolicyCenter provides the following *internal* (non-General) activity patterns.

- Approval
- Approval Denied
- Assignment Review

Any pre-existing activity patterns of type *General* in the base configuration are examples that Guidewire provides. You can fully customize any of them. Activity patterns with other types are typically not available in the PolicyCenter interface. You use them only within Gosu and PolicyCenter uses them internally.

### Categorizing Activity Patterns

Guidewire recommends that you categorize your activity patterns so that it is possible to choose among the different activity categories during new activity creation. These categories serve as the first level of navigation in the PolicyCenter **New Activity** menu. The activity pattern categories appear only within the PolicyCenter interface.

### The ActivityCategory Typelist

Guidewire defines activity categories in the `ActivityCategory` typelist. You are free to add or delete typecodes from this typelist. If you change a typelist, remember that you must restart the application server to view your changes in the PolicyCenter interface.

PolicyCenter displays the activity categories in the [New Activity Pattern](#) editor screen.

## Using Activity Patterns in Gosu

**IMPORTANT** You *must* use the activity pattern code to refer to an activity pattern in Gosu code. Do not use a pattern ID or PublicID value.

There are two operations that you can perform in Gosu involving activity patterns:

- One is to test which activity pattern an existing activity uses.
- The other is to retrieve an activity pattern for use in creating a new activity.

### To test for a specific activity pattern

Use the following Gosu code, which compares an activity pattern `Code` value with a string value that you supply.

```
Entity.ActivityPattern.Code == "activity_pattern_code"
```

### To retrieve an activity pattern

To find (retrieve) a specific activity pattern, use one of the following Gosu `find` or `get` methods. The `find` method returns a query object and the `get` method returns an `ActivityPattern` object.

```
ActivityPattern.finder.findActivityPatternsByCode("activity_pattern_code")
ActivityPattern.finder.getActivityPatternByCode("activity_pattern_code")
```

Any query object that the `find` method returns exists in its own read-only bundle separate from the active read-write bundle of any running code. To change the properties on a read-only entity, you must move (add) the entity to a new writable bundle. From more information, see “[Updating Entity Instances in Query Results](#)” on page 172 in the *Gosu Reference Guide*.

To create an activity based on a specific activity pattern, use the following Gosu code. Notice the use of the embedded `get` method to retrieve the correct `ActivityPattern` object.

```
Entity.createActivityFromPattern(null,
 ActivityPattern.finder.getActivityPatternByCode("activity_pattern_code"))
```

### See also

- “[Use Activity Pattern Codes Instead of Public IDs in Comparisons](#)” on page 39 in the *Best Practices Guide*

## Calculating Activity Due Dates

The activity made from a pattern always has a specific date as a deadline. Each activity pattern defines how to calculate the due date for a specific activity instance.

## Target Due Dates (Deadlines)

A *target date* (or *due date*) suggests the date to complete an activity. Settings in the **New Activity Pattern** editor determine how PolicyCenter calculates the due date for an activity. PolicyCenter can calculate a target due date in hours or days. PolicyCenter calculates due dates using the following pieces of information:

- *How much time?* How much time to take or how many hours or days to allow to complete the activity. You specify this using the **Target days** or **Target hours** value.
- *What is the starting point?* What point in time does PolicyCenter use as the start point in calculating the target date? You specify this using the **Target start point** field.
- *What days to count?* PolicyCenter can count calendar days or only business days. You specify this with the **Target Include Days** field.

PolicyCenter reports deadlines only at the level of days. For example, if something is due on 6/1/2008, it becomes overdue on 6/2/2008, not some time in the middle of the day on 6/1. PolicyCenter does track activity creation dates and marks completion at the level of seconds so that you can calculate average completion times at a more granular level.

If you do not specify **Target Days** or **Target Hours** as you define an **Activity Pattern Detail**, PolicyCenter uses 0 for both. A target date is optional for activities.

## Escalation Dates

While the target date can indicate a service-level target (for example, complete within five business days), there can possibly be some later deadline after which the work becomes dangerously late. (This can be, for example, a 30 day state deadline.) PolicyCenter calls this later deadline an escalation date.

The escalation date is the date at which activity requires urgent attention. While work is shown as overdue after the target date, PolicyCenter does not actually escalate (take action on) an activity until the escalation date passes. Within Studio, you can define a set of rules that define what actions take place if an activity reaches its escalation date. For example, it could be company policy to inform a supervisor if an activity passes an escalation date. You might also want to reassign the activity.

PolicyCenter calculates the escalation date using the methodology it uses for target dates. You can specify escalation timing in days and hours. If you do not specify **Escalation Days** or **Escalation Hours** as you define an activity pattern, PolicyCenter uses 0 (zero) for both. An escalation date, like a target date, is optional for activities.

## Configuring Activity Patterns

PolicyCenter uses file `activity-patterns.csv` to load the base activity pattern definitions upon initial server startup after installation. You can customize the activity patterns in the `activity-patterns.csv` file and re-import them. Or, you can customize them through the PolicyCenter **Administration** tab. You can access the `activity-patterns.csv` file through Guidewire Studio by navigating to the `configuration → config → import → gen` folder.

---

**IMPORTANT** Do not remove any internal (non-General type) activity patterns or change their type, category, or code values. Internal PolicyCenter application code requires them. You can change other fields associated with these types, however.

---

The **ActivityPattern** object contains the following properties:

Property	User interface field	Description
ActivityClass	Activity Class	Indicates whether the activity is a task or an event. A task has a due date. An event does not.
AutomatedOnly	Automated only	A Boolean value that defines whether only automated additions (by business rules) to the workplan use the activity pattern. <ul style="list-style-type: none"> <li>• If true, the activity pattern does not appear as a choice in PolicyCenter interface.</li> <li>• If you do not specify this value, the default is false.</li> </ul> <p>Guidewire recommends that you set this flag set to true for all patterns with a non-general type. This ensures that they are not visible in the PolicyCenter interface.</p>
Category	Category	The category for grouping <b>ActivityPatterns</b> in the PolicyCenter interface.
Code	Code	Any unique text <i>with no spaces</i> . Maximum length is 60 characters. This property is required. The Code property is used to identify the activity pattern when accessing the pattern in rules or Gosu code. You can see this value only through the <b>Administration</b> tab.
Command	Not Applicable	<i>Do not use.</i> For Guidewire use only.
Data-Set	Not Applicable	The value of the highest-numbered data set of which the imported object is a part. PolicyCenter typically orders a data set by inclusion. Thus, data set 0 is a subset of data-set 1, and data set 1 is a subset of data set 2, and so forth.
Description	Description	Describes the expected outcome at the completion of this activity. It is visible only if you view the details of the activity.
DocumentTemplate	Document Template	Document template to display if you choose this activity. Enter the document template ID.
EmailTemplate	Email Template	Email template to display if you choose this activity. Enter the email template file name.
EntityId	Not Applicable	<i>Required.</i> The unique public ID of the activity pattern.
EscalationDays	Escalation days	The number of days from the escalationstartpt to set the Escalation Date for an activity.
EscalationHours	Escalation hours	The number of hours from the escalationstartpt to set the Escalation Date for an activity.
EscalationInclDays	Escalation Include Days	Specifies which days to include. You can set this businessdays or elapsed.
EscalationStartPt	Escalation start point	The initial date used to calculate the target date. If you specify escalationdays or escalationhours, you need to specify this parameter. Otherwise, this parameter is optional.
Mandatory	Mandatory	A Boolean value that defines whether you can skip an activity. Non-mandatory activities act as suggestions about what might be a useful task without forcing you into doing unnecessary work. This value is optional. If you do not specify a value, the application uses a default of true.
PatternLevel	Pattern Level	The level that this pattern is for. Valid choices are: <ul style="list-style-type: none"> <li>• Account</li> <li>• All</li> <li>• Job</li> </ul>
Priority	Priority	Used to sort more important activities to the top of a list of work. This property is required. You can set this property to the following values: <ul style="list-style-type: none"> <li>• urgent</li> <li>• high</li> <li>• normal</li> <li>• low.</li> </ul>

Property	User interface field	Description
Recurring	Recurring	A Boolean value indicating that an activity is likely to recur on a regular schedule. If you do not specify a value, the application uses a default of true.
ShortSubject	Short Subject	A brief description of the activity used on small areas of the PolicyCenter interface such as a calendar event entry. Maximum length of 10 characters.
Subject	Subject	A short text description of the activity that PolicyCenter shows in activity lists. This property is required.
TargetDays	Target days	The number of days from the targetstartpoint to set the activity's Target Date.
TargetHours	Target hours	The number of hours from the targetstartpoint to set the activity's Target Date.
TargetIncludeDays	Target Include Days	This field answers the "what days to count" part of calculating the target date. Your options are the following: <ul style="list-style-type: none"> <li>• elapsed—the count all days</li> <li>• businessdays—as defined by the business calendar</li> </ul>
TargetStartPoint	Target start point	The initial date used to calculate the target date. You need specify this value only if you specify targetdays or targethours. Otherwise, this value is optional.
Type	Type	This specifies what activity type to create. You must use the <i>General</i> pattern for all your custom activities.

## Using Activity Patterns with Documents and Emails

It is possible to attach a specific document or email template to a specific activity pattern. Then, as PolicyCenter displays an activity based on this activity pattern, it displays a **Create Document** or **Create Email** button in the **Activity Detail** worksheet. This indicates that this type of activity usually has a document or email associated with that activity.

### To associate a document or email template with an activity pattern.

1. Log into Guidewire PolicyCenter under an administrative account and access the following screen:

Administration → Activity Patterns

2. Open the activity pattern edit screen by either creating a new activity pattern or selecting an activity pattern to update.

---

Create new → Click **Add Activity Pattern**

Update existing → Select an activity pattern and click **Edit**

---

3. Use the spyglass icon next to the **Document Template** and **Email Template** fields to open a search window.

4. Find the desired document or email template, then add it to the activity pattern.

If you associate a document or email template with an activity pattern, PolicyCenter does the following:

- If you create a new activity from this activity pattern, PolicyCenter automatically populates any template field for which you specified a template with the name of that template.
- If you then open this activity, PolicyCenter displays a **Create Document** and a **Create Email** button in the **Activity Detail** worksheet at the bottom of the screen. (That is, if you specified a template for each type in the activity pattern.)

- If you then click the **Create Document** or the **Create Email** button, PolicyCenter creates the document or email and populates its fields according to the specified template.

**Note:** You can also specify the document or email template in file `activity-patterns.csv`. Add a column for that template and then enter either the document template ID or the email template file name as appropriate. See “Configuring Activity Patterns” on page 408 for details of working with the `activity-patterns.csv` file.

## Localizing Activity Patterns

PolicyCenter stores activity pattern data directly in the database. Thus, it is not possible to localize fields such as the subject or description of an activity pattern by localizing a display string. In the base configuration, you can localize the following activity pattern properties (fields) through the PolicyCenter interface—if you configure PolicyCenter for multiple locales:

- Description
- Subject

If you configure PolicyCenter correctly to use multiple locales, then you see additional fields at the bottom of the **New Activity Pattern** screen. You use these fields to enter localized subject and description text for that activity pattern.

### See also

- For information on how to make a database column localizable (and thus, an object property localizable), see “Localizing Administration Data” on page 61 in the *Globalization Guide*.



# Testing Gosu Code



# Testing and Debugging Your Configuration

After you use Guidewire Studio to make configuration changes to your application, you will typically want to run the application to test those changes. Guidewire Studio provides powerful features to help you make sure that your application works the way you intend.

This topic includes:

- “Testing PolicyCenter With Guidewire Studio” on page 415
- “The Studio Debugger” on page 418
- “Setting Breakpoints” on page 418
- “Stepping Through Code” on page 419
- “Viewing Current Values” on page 420
- “Resuming Execution” on page 421
- “Using the Gosu Scratchpad” on page 421
- “Suggestions for Testing Rules” on page 422

## Testing PolicyCenter With Guidewire Studio

After you make configuration changes to PolicyCenter, you can start it directly from within Guidewire Studio and test your changes. You can also use the powerful debugging features that Studio provides.

This topic contains:

- “Running PolicyCenter Without Debugging” on page 416
- “Debugging PolicyCenter Within Studio” on page 416
- “Debugging a PolicyCenter Server That Is Running Outside of Studio” on page 416

## Running PolicyCenter Without Debugging

If you do not plan to use debugging features, then you can run PolicyCenter directly from within Studio to quickly test your configuration changes. Running PolicyCenter this way is similar to starting it from the command line with the command `gwpc dev-start`, but without having to switch out of Studio.

### To run PolicyCenter without debugging

- In Studio, click Run → Run 'Server'.

The PolicyCenter server starts, and debug messages appear in the Run tool window in Studio.

## Debugging PolicyCenter Within Studio

You can run PolicyCenter in the Studio debugger, which provides additional features to help you verify that your configuration changes are working as desired.

### To debug PolicyCenter within Studio

- In Studio, click Run → Debug 'Server'.

The PolicyCenter server starts, and debug messages appear in the Debug tool window in Studio.

#### See also

- “The Studio Debugger” on page 418

## Debugging a PolicyCenter Server That Is Running Outside of Studio

Instead of running PolicyCenter within the Studio debugger, you can have the debugger connect to a PolicyCenter server that is running outside of Studio. The debugger can connect to a PolicyCenter server running either on the local computer or on a remote computer.

You must choose one of the following ways for Studio to connect to the server:

Connection type	Description	See
shared memory	Connects Studio to a server running on the same computer; faster than a socket connection.	“Debugging a PolicyCenter Server Using a Shared Memory Connection” on page 416
socket	Allows Studio to connect to a server running on a remote computer; slower than a shared memory connection.	“Debugging a PolicyCenter Server Using a Socket Connection” on page 417

### Debugging a PolicyCenter Server Using a Shared Memory Connection

Studio can use a shared memory connection to connect to a PolicyCenter server running on the same computer. You must manually start the server in the proper debug mode, and create the proper debug configuration in Studio.

1. Start the PolicyCenter server.

a. Start the server using the following command:

```
gwpc dev-debug-shmem
```

b. Towards the beginning of the server console message output, look for the label `dev-debug-shmem:`, and then the text that is similar to the following.

```
Listening for transport dt_shmem at address: javaAddress
```

c. Make a note of the value provided for `javaAddress`.

2. Create the debug configuration in Studio.
  - a. In Studio, click Run → Edit Configurations.
  - b. In the Run/Debug Configurations dialog, click Add New Configuration , and then click Remote.
  - c. In the Name text box, type a name for the configuration. For example, server-shmem.
  - d. For the Transport option, click Shared memory.
  - e. In the Shared Memory Address text box, type the *javaAddress* that you noted when you started the server.
  - f. Click OK.
3. Connect the Studio debugger to the PolicyCenter server using the shared memory connection.
  - a. In Studio, in the Select Run/Debug Configuration drop-down list, select the debug configuration that you created.
  - b. Click Run → Debug '*configName*', where *configName* is the name of the debug configuration that you created.

The debugger connects to the PolicyCenter server, and debug messages appear in the Debug tool window in Studio.

### Debugging a PolicyCenter Server Using a Socket Connection

Studio can use a socket connection to connect to a PolicyCenter server running either on the same computer or a remote computer. You must manually start the server in the proper debug mode, and create the proper debug configuration in Studio.

1. Start the PolicyCenter server.
  - a. Start the server using the following command:  
`gwpc dev-debug-socket`
  - b. Towards the beginning of the server console message output, look for the label `dev-debug-socket:`, and then the text that is similar to the following.  
`Listening for transport dt_socket at address: portNumber`
  - c. Make a note of the value provided for *portNumber*.
2. Create the debug configuration in Studio.
  - a. In Studio, click Run → Edit Configurations.
  - b. In the Run/Debug Configurations dialog, click Add New Configuration , and then click Remote.
  - c. In the Name text box, type a name for the configuration. For example, server-socket.
  - d. For the Transport option, click Socket.
  - e. In the Host text box, type the hostname of the computer on which the PolicyCenter server is running.
  - f. In the Port text box, type the *portNumber* that you noted when you started the server.
  - g. Click OK.
3. Connect the Studio debugger to the PolicyCenter server using the shared memory connection.
  - a. In Studio, in the Select Run/Debug Configuration drop-down list, select the debug configuration that you created.
  - b. Click Run → Debug '*configName*', where *configName* is the name of the debug configuration that you created.

The debugger connects to the PolicyCenter server, and debug messages appear in the Debug tool window in Studio.

## Debugging a PolicyCenter Server in Suspended Mode

When you start a PolicyCenter server in debug mode, the server completes its full startup process until it becomes ready for client connections. In this case, you cannot begin to debug the server until it is ready. However, if the behavior that you want to debug occurs earlier in the startup process, then you can start the server in *suspended* mode instead. In suspended mode, the PolicyCenter startup pauses as soon as its Java process is initially established. The startup process continues only once you start the Studio debugger.

To start the PolicyCenter server in suspended mode, use one the following commands instead of the debug commands:

- `gwpc dev-suspend-shmem`
- `gwpc dev-suspend-socket`

## The Studio Debugger

Guidewire Studio includes a code *debugger* to help you verify that your Gosu code is working as desired. It works whether the code is in a Gosu rule, a Gosu class, or a PolicyCenter PCF page. You access this functionality through the Studio **Run** menu and through specific debug icons on the Studio toolbar. You must be connected to a running PolicyCenter server to use the Studio debugger. (If you do not have a connection to a running server, Studio attempts to run one.) If the debugger is active, you can debug Gosu code that runs in the Gosu Scratchpad and Gosu code that is part of the running application.

If instructed, Studio can pause (at a breakpoint that you set) before it runs a specified line of code. This can be any Gosu code, whether contained in a rule or a Gosu class. The debugger can also run on Gosu that you call from a PCF page, if the called code is a Studio class.

After Studio pauses, you can examine any variables or properties used by Gosu and view their values at that point in the debugger pane. You can then have Studio continue to step through your code, pausing before each line. This allows you to monitor values as they change, or simply to observe the execution path through your code.

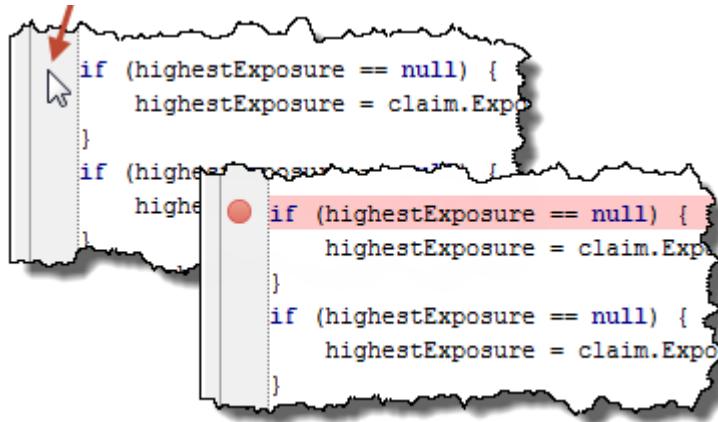
**Note:** Do not perform debugging operations on a live production server.

## Setting Breakpoints

A breakpoint is a place in your code at which the debugger pauses execution, giving you the opportunity to examine current values or begin stepping through each line. The debugger pauses before executing the line containing the breakpoint. The debugger identifies a breakpoint by highlighting the related line of code and placing a breakpoint symbol  next to it.

### To set a breakpoint

Place the cursor on the line of code on which to set the breakpoint, and then on the Run menu, click **Toggle Line Breakpoint**. You can also click in the gray column next to a code line:



You can set multiple breakpoints throughout your code, with multiple breakpoints in the same block of code, or if desired, breakpoints in multiple code blocks. The debugger pauses at the first breakpoint encountered during code execution. After it pauses, the debugger ignores other breakpoints until you continue normal execution.

You can set a breakpoint in a rule condition statement, as well. You cannot set a breakpoint on a comment.

### To view a breakpoint

On the Run menu, click **View Breakpoints**. Selecting this menu item opens that **View Breakpoints** dialog in which you can do the following:

- View all of your currently set breakpoints
- Deactivate any or all of your breakpoints (which makes them non-functional, but does not remove them from the code)
- Remove any or all breakpoints
- Navigate to the code that contains the breakpoint

Thus, from this dialog, you can deactivate, remove, or add a breakpoint.

### To remove a breakpoint

Place the cursor on the line of code containing the breakpoint to remove, and then on the Run menu, click **Toggle Line Breakpoint**. You can also click the breakpoint symbol next to the line of code.

## Stepping Through Code

After the debugger pauses execution, you can step through the code one line at a time in one of the following ways:

<b>Step over</b>	Execute the current line. If the current line is a method call, then run the method and return to the next line in the current code block after the method ends. To step through your code in this manner, on the Run menu, click <b>Step Over</b>  .
<b>Step into</b>	Execute the current line of code. If the current line is a method call, then step into the method and pause before executing the first line for that method. To step through your code in this manner, on the Run menu, click <b>Step Into</b>  .

The only difference between these two stepping options is if the current line of code is a method call. You can either *step over* the method and let it run without interruption, or you can *step into* it and pause before each line. For other lines of code that are not methods, stepping over and stepping into behave in the same way.

While paused, you can navigate to other rules or classes if you want to look at their code. To return to viewing the current execution point, on the Run menu, click **Show Execution Point** . Studio highlights the line of code to execute the at the next step.

## Viewing Current Values

After the debugger pauses at a breakpoint in your code, you can examine the current values of variables or entity properties. You can watch these values and see how they change as each line of code executes.

This topic includes:

- “Enabling the Viewing of Entities While Debugging” on page 420
- “Viewing Variables” on page 420
- “Defining a Watch List” on page 420

### Enabling the Viewing of Entities While Debugging

Collecting and displaying real-time entity information can make the server run slower, so you can turn this feature on and off as needed.

#### To enable the viewing of entities

1. In Guidewire Studio, click .
2. Navigate to the Guidewire Studio page, and then select **Enhance Entities Visualization**.
3. When the server pauses at a breakpoint, look in the **Debug** pane, and then in the **Variables** or **Watches** list. Locate the entity that you want to view.
4. Right-click the entity, and then click **View as → Entity**.

### Viewing Variables

The **Debugger** tab of the **Debug** pane shows the root entity that is currently available. For example, in activity assignment code, the root entity is an **Activity** object. To view any property of the root entity, expand it until you locate the property.

The **Debugger** tab also shows the variables that you have currently defined. Note, however, that you can view only the entities and variables that are available in the current *scope* of the code. Suppose, for example, that an **Activity** entity is available in an activity assignment class. However, if that class calls a different class and you step into that class, the **Activity** entity is no longer part of the scope. Therefore, it is no longer available. (Unless, you pass the **Activity** object in as a parameter.)

### Defining a Watch List

After executing each line of code, Studio resets the list of values shown in the **Debug** frame. In doing so, it collapses any property hierarchies that you may have expanded. It would be inconvenient for you to expand the hierarchy after each line and locate the desired properties all over again.

Instead, you can define a watch list containing the Gosu expressions in which you have an interest in monitoring. The debugger evaluates each expression on the list after each line of code executes, and shows the result for each expression on the list. For example, you can add `Activity.AssignedUser` to the watch list and then monitor the list as you step through each line of code. If the property changes, you see that change reflected immediately on the list. You can also add variables to the list so you can monitor their values, as well.

To add an expression to the watch list, in the **Variables** pane, right-click the expression, and then click **Add to Watches**.

As you step through each line of code in the debugger, keep the **Watches** pane visible so that you can monitor the values of the items on the list.

## Resuming Execution

After the debugger pauses execution of your code, and after you are through performing any necessary debugging steps, you may want to resume normal execution. You can do this in the following ways:

<b>Stop debugging</b>	Resume normal execution, and ignore all remaining breakpoints. To stop debugging, click <b>Mute Breakpoints</b>  , and then click <b>Resume Execution</b>  .
<b>Continue debugging</b>	Resume normal execution, but pause again at the next breakpoint, if any. To continue debugging, click <b>Resume Execution</b>  without having <b>Mute Breakpoints</b>  set.

## Using the Gosu Scratchpad

You use the Gosu Scratchpad to execute Gosu programs, evaluate Gosu expressions, and process Gosu templates. Instead of needing to perform PolicyCenter operations to trigger your Gosu code, you can run your code directly in the Scratchpad and see immediate results.

To run the Gosu Scratchpad, click **Gosu Scratchpad**  . It is possible to save a Gosu test expression and open it again in the Gosu Scratchpad by using the appropriate menu commands.

To execute queries against the database in Studio or the Gosu tester, you must first connect to a running application server. The result of a query is always a read-only query object. To work with this read-only object, you must add it to a transaction bundle. See also “Using the Results of Find Expressions (Using Query Objects)” on page 187 in the *Gosu Reference Guide*.

The Gosu Scratchpad has the following modes:

Mode	Description
Expression	Returns the result of evaluating the (single-line) expression.
Program	Displays the output of the program including calls to <code>print</code> and exception stack traces.
Template	Displays the resulting content from executing the template.

**Note:** If you create code that contains an infinite loop in the Gosu Scratchpad, it is not sufficient to shut down the Scratchpad to correct the problem. You must also shut down and restart Studio itself. Merely shutting down the Gosu Scratchpad is not sufficient to stop the running JVM.

## Testing a Gosu Expression

To test an expression in the Gosu Scratchpad, type your expression, and then click **Run**  .

## Suggestions for Testing Rules

Guidewire recommends that you practice the following simple suggestions to make testing and debugging your rules a straightforward process:

- Enter one rule at a time and monitor for syntax correctness—check the green light (at the bottom of the pane) before starting a new rule.
- Enter rules in the order in which you want the debugger to evaluate them: **Condition** and then **Action**.
- Maintain two sessions while testing. As you complete and save each rule in Studio, toggle to an open PolicyCenter session and test before continuing. You only need save and activate your rules before testing. You do not need to log in again.

For multi-conditioned rules, you can print messages to the console after each action for easy monitoring. The command for this is `print("message text")`. The message prints in the server console. This is helpful if you want to test complex rules and verify that Studio evaluated each case.

Other print-type statements that you can use for testing and debugging include the following:

```
gw.api.util.Logger.logDebug
gw.api.util.Logger.LogError
gw.api.util.Logger.logInfo
gw.api.util.Logger.logTrace
```

These all log messages as specified by the PolicyCenter logging settings.

# Using GUnit

You use Studio GUnit to configure and run repeatable tests of your Gosu code in a similar fashion as JUnit works with Java code. (GUnit is similar to JUnit 3.0 and compatible with it.) GUnit works automatically and seamlessly with the embedded QuickStart servlet container, enabling you to see the results of your GUnit Gosu tests within Studio.

GUnit provides a complete test harness with base classes and utility methods. You can use GUnit to test any body of Gosu code except for Gosu written as part of Rules. (To test Gosu in Rules, use the Studio debugger. See “Testing and Debugging Your Configuration” on page 415 for details.)

This topic includes:

- “The TestBase Class” on page 423
- “Configuring the Server Environment” on page 424
- “Configuring the Test Environment” on page 426
- “Creating a GUnit Test Class” on page 428
- “Using Entity Builders to Create Test Data” on page 430

**Note:** Guidewire does not recommend or support the use of classes that extend `gw.api.databuilder.DataBuilder` or classes that reside in the `gw.api.databuilder.*` package in a production environment. Guidewire provides GUnit as a development test facility only.

## The TestBase Class

Guidewire uses the `TestBase` class as the root class for all GUnit tests. Your test class must extend the Guidewire `TestBase` class. This class provides the following:

- The base test infrastructure, setting up the environment in which the test runs.
- A set of `assert` methods that you can use to verify the expected result of a test.
- A set of `beforeXX` and `afterXX` methods that you can override to provide additional testing functionality (for example, to set up required data before running a test method).

The `TestBase` class interacts with an embedded QuickStart servlet container in running your GUnit tests. This class has access to all of the embedded QuickStart server files and servlets. (GUnit starts and stops the embedded QuickStart servlet container automatically. You have no control over it.) This class also initializes all server dependencies.

## Overriding TestBase Methods

Guidewire exposes two groups of `beforeXX` and `afterXX` methods in the `TestBase` class that you can use to perform certain actions before and after the tests execute. These methods are a way to set up any required dependencies for tests and to clean up after a test finishes.

To use one of these methods, you need to provide an overridden implementation of the method in your test class.

- Use `beforeClass` to perform some action before GUnit instantiates the test class.
- Use `afterClass` to perform some action after all the tests complete but before GUnit destroys the class.
- Use `beforeMethod` to perform some action before GUnit invokes a particular test method.
- Use `afterMethod` to perform some action after a test method returns.

These methods have the following signatures.

```
beforeClass() throws Exception {...}
afterClass() {...}
beforeMethod() throws Exception {...}
afterMethod(Throwable possibleException) {...} //If the test resulted in an exception, parameter
//possibleException contains the exception.
```

### Data Builders

If you need to set up test data before running a test, Guidewire recommends that you use a “data builder” in one of the `beforeXX` methods.

- See “Using Entity Builders to Create Test Data” on page 430 for details on how to create test data.
- See “Creating a Builder for a Custom Entity and Testing It” on page 437 for details of using the `beforeClass` method to create test data before running a test.

## Configuring the Server Environment

Annotations control the way GUnit interacts with the system being tested. There are two types of annotations:

Annotation type	Description
Server Runtime	This annotation indicates that this test interacts with the server.
Server Environment	These can provide additional test functionality. Use them to replace or modify the default behavior of the system being tested.

To use an annotation, either enter the full path:

```
@gw.testharness.ServerTest
```

Or, you can add a `uses` statement at the beginning of the file, for example:

```
uses gw.testharness
...
@ServerTest
```

## Server Runtime

A server test is a test written in the environment of a running server. The test and the server exist in the same JVM (Java Virtual Machine) and in the same class loader. This allows the test to communicate with the server using standard variables. In the base configuration, Guidewire uses an embedded QuickStart servlet container pointing at a Web application to run the tests.

PolicyCenter interprets any class that contains the annotation `@ServerTest` immediately before the class definition as a server test. If you create a test class through Guidewire Studio, then Studio automatically adds the server runtime annotation `@ServerTest` immediately before the class definition. At the same time, Studio also adds `extends gw.testharness.TestBase` to the class definition. All GUnit tests that you create must extend this class. (See the “The TestBase Class” on page 423 for more information on this class.)

Although Studio automatically adds the `@ServerTest` annotation to the class definition, it is possible to remove this annotation safely. As the `TestBase` class already includes this annotation, Guidewire does not explicitly require this annotation in any class that extends the `TestBase` class.

By default, the server starts at a run level set to `RunLevel.NO_DAEMONS`. To change this default, see the description of the `@RunLevel` annotation in the next section.

## Server Environment

Environment tags provide additional functionality. You use environment tags to replace functionality specific to an external environment. This can include defining new SOAP endpoints or creating tests for custom PCF page, for example.

Guidewire provides the following environment tags for use in GUnit tests.

Annotation (@gw.testharness.*)	Description
<code>@ChangesCurrentTime</code>	Sets up a mock system clock that allows the test to change the current time during the test.
<code>@ProductUnderTest</code>	Explicitly sets the product being tested. Typically, Studio infers this from the test class package. However, you can use this annotation if that is not possible, as with <code>gw.api</code> tests, for example.
<code>@ProductionMode</code>	GUnit runs tests against the QuickStart servlet container, by default, in “development” mode. If desired, you can direct GUnit to run tests against the QuickStart servlet container in “production” mode, which duplicates the system functionality available to a running production application server. If you do so, you may lose test functionality that is only available in development mode (for example, access to the system clock).  You can check the server mode in Gosu, using the following: <code>gw.api.system.server.ServerModeUtil.isDev()</code>
<code>@RealPCFs</code>	Loads the production PCF files for the application. If you do not include this annotation, Studio does not load the PCF files. This reduces the amount of time needed for startup.

Annotation (@gw.testharness.*)	Description
@RunInDatabase	Defines the databases against which to run this class's tests. Without this annotation, this class only runs in H2 suites. The annotation takes an array of DatabaseForTest values, specifying the databases which are specifically to be tested, or DatabaseForTest.ALL that allows the class to be run against any database.
@RunLevel	Allows a test to run at a different run level. The default value is Runlevel.NO_DAEMONS. You can, however, change the run level to one of the following (although each level takes a bit more time to set up): <ul style="list-style-type: none"> <li>• Runlevel.NONE - Use if you do not want any dependencies at all.</li> <li>• Runlevel.SHUTDOWN - Use if you want all the basic dependencies set up, but with no database connection support.</li> <li>• Runlevel.NO_DAEMONS - Use for a normal server startup without background tasks. (This is also suitable for SOAP tests.)</li> <li>• Runlevel.MULTIUSER - Use to start a complete server (batch process, events, rules, Web requests, and all similar components).</li> </ul>

## Configuring the Test Environment

You define the run and debug parameter settings for a GUnit test class through the **Run/Debug Settings** dialog, which you can access in any of the following ways:

- Click **Run** → **Edit Configurations**.
- On the main toolbar, in the **Select Run/Debug Configuration** drop-down list, click **Edit Configurations**.
- In the **Project** tool window, right-click the test package, and then click **Create 'Tests in 'PackageName'**.
- Open the test class in the editor, right-click anywhere in the method, and then click **Create 'testName()'**.

You can set various default configuration parameters for all tests, or configure parameters for a particular test.

### Setting Default Configuration Parameters for All Tests

It is possible to set a number of default configuration parameters that GUnit uses for all tests. To do this, in the **Run/Debug Configurations** dialog, expand **Defaults**, and then click **Junit**. Enter the default configuration parameters as appropriate. See “Configuration Parameters” on page 427 for a description of the various configuration parameters.

### Adding a Named Set of Configuration Parameters

It is possible to create a defined set of configuration parameters to use with one or more tests. To do this, first add that configuration under the **Application** section of the list in the **Run/Debug Configurations** dialog. Use the following dialog toolbar icons.

Icon	Use to
	Add a new named test configuration to the list. Click this, and then click <b>JUnit</b> .
	Delete the selected configuration from the list.
	Clone the selected test configuration.
	Move the selected configuration up within the list.
	Move the selected configuration down within the list.

## Viewing Configuration Settings Before Launching

It is possible to turn on, or off, the Run/Debug Configurations dialog before running a test. To view the GUnit configuration settings before launching a test, expand the **Before launch** section of that dialog, and then set the **Show this page** check box.

If you unset this option, you do not see the Run/Debug Configurations dialog upon starting a test. Instead, the test starts immediately. In addition, selecting **Run** or **Debug** from the Studio **Run** menu does not open this dialog either. To access the Run/Debug Configurations dialog again, click **Run → Edit Configurations**.

## Configuration Parameters

Use the Run/Debug Configurations dialog to enter the following configuration parameters:

- Name
- Test Kind
- VM Options

You may not see some of the parameter fields until you actually load a test into Studio and select it. See “Creating a GUnit Test Class” on page 428 for information on how to create a GUnit test within Guidewire Studio.

### Name

If desired, you can set up multiple run and debug GUnit configurations. Each named configuration represents a different set of run and debug startup properties. To create a new named configuration:

- Click **Add**  and create a new blank configuration.
- Select an existing configuration, then click **Copy Configuration**  to copy the existing configuration parameters to the new configuration.
- Select the test class in the Project window, and then click either **Run 'TestName'** or **Debug 'TestName'**. Then, select the name of the test from the list of GUnit tests and click **OK**. This has an advantage of populating the fully qualified class name field.

After you add the new configuration node on the left-hand side, you can enter a name for it on the right-hand side of the dialog.

### Test Kind

Use to set whether to test all the classes in a package, a specific class, or a specific method in a class. The text entry field changes as you make your selection.

- For **All in Package**, enter the fully qualified package name. Select this option to run all GUnit tests in the named package.
- For **Class**, enter the fully qualified class name. Select this option to run all GUnit tests in the named class.
- For **Method**, enter both the fully qualified class name and the specific method to test in that class.

### VM Options

Use to set parameters associated with the JVM and the Java debugger. To set specific parameters for the JVM to use while running this configuration, enter them as a space separated list in the **VM Options** text box. For example:

```
-client -Xmx700m -Xms200m -XX:MaxPermSize=100m -ea
```

You can change the JVM parameters based on the test. For example, while testing a large class or while running numerous test methods within a class, you may want to increase your maximum heap size.

## Creating a GUnit Test Class

The following is an example of a GUnit test class. Use this sample code as a template in creating your own test classes.

```
package AllMyTests

uses gw.testharness.TestBase
@gw.testharness.ServerTest
class MyTest extends TestBase {

 construct(testname : String) {
 super(testname)
 }
 ...
 function testSomething() {
 //perform some test
 assertEquals("reason for failure", someValue, someOtherValue)
 }
 ...
}
```

Notice the following:

- The test class exists in the package `AllMyTests`. Thus, the full class path is `Tests.AllMyTests.MyTest`. You must place your test classes in the `modules/configuration/gtest` folder. You are free, however, to name your test subpackages as you choose.
- The class file name and the class name are identical and end in `Test`.
- The test class extends `TestBase`.
- The class definition files contains a `@ServerTest` annotation immediately before the class definition.
- The class definition contains a `construct` code block. This code block can be empty or it may contain initialization code.
- The class definition contains one or more test methods that begin with the word `test`. The word `test` is case-sensitive. For example, GUnit will recognize the string `testMe` as a method name, but not the string `TestMe`.
- The test method contains one or more `assert` methods, each of which “asserts” an expected result on the object under test.

### Server Tests

You specify the type of test using annotations. Currently, Guidewire supports server tests only. Server tests provide all of the functionality of a running server. You must include the `@ServerTest` annotation immediately before the test class definition to specify that the test is a server test. See “Configuring the Server Environment” on page 424 for more information on annotations.

### The Construct Block

Gosu calls the special `construct` method if you create a new test using the `new Object` construction. For example:

```
construct(testname : String) {
 super(testname)
}
```

This `construct` code block can be empty or it may contain initialization code.

### Test Methods

Within your test class, you need to define one or more test methods. Each test method must begin with the word `test`. (GUnit recognizes a method as test method only if the method name begins with `test`. If you do not have at least one method so named, GUnit generates an error.) Each test method uses a verification method to test a

single condition. For example, a method can test if the result of some operation is equal to a specific value. In the base configuration, Guidewire provides a number of these verification methods. For example:

- `assertTrue`
- `assertEquals`
- `verifyTextInPage`
- `verifyExists`
- `verifyNull`
- `verifyNotNull`

Many of these methods appear in multiple forms. Although there are too many to list in their entirety, the following are some of the basic assert methods. To see a complete list of these methods in their many forms, use the code completion feature in Studio.

```
assertArrayDoesNotContain
assertArrayEquals
assertBigDecimalEquals
assertBigDecimalNotEquals
assertCollection
assertCollectionContains
assertCollectionDoesNotContain
assertCollectionContains
assertCollectionSame
assertComparesEqual
assertDateEquals
assertEmpty
assertEquals
assertEqualsIgnoreCase
assertEqualsIgnoreLineEnding
assertEqualsUnordered
assertFalse
assertFalseFor
assertGreaterThan
assertIteratorEquals
assertIteratorSame
assertLength
assertList
assertListEquals
assertListSame
assertMethodDeclaredAndOverridesBaseClass
assertNotNull
assertNotSame
assertNotZero
assertNull
assertSame
assertSet
assertSize
assertSuiteTornDown
assertThat
assertTrue
assertTrueWithin
assertZero
```

### The `assertThat` Method

Choosing the `assertThat` method opens up a whole variety of different types of assertions, dealing with strings, collections, and many other object types. To see a complete list of this method in its many forms, use the code completion feature in Studio.

### Failure Reasons for Asserts

Guidewire strongly recommends that, as appropriate, you use an assert method that takes a string as its first parameter. For example, even though Guidewire supports both versions of the following assert method, the second version is preferable as it includes a failure reason.

```
assertEquals(a, b)
assertEquals("reason for failure", a, b)
```

Guidewire recommends that you document a failure reason as part of the method rather than adding the reason in a comment. The GUnit test console displays this text string if the assert fails, which makes it easier to understand the reason of a failure.

#### To create a GUnit test class

1. In the Project window, navigate to configuration → gtest.
1. Right-click gtest, and then click New → Package.
2. In the Enter new package name text box, type the name of the package.
3. Right-click the new package, and then click New → Gosu Class.
4. In the Name text box, type the name of the test class. This class file name must match the test class name and both must end in “Test”. This action creates a class file containing a “stub” class. For example, if your class file is MyTest.gs, Studio populates the file with the following Gosu:

```
package demo

@gw.testharness.ServerTest
class MyTest extends gw.testharness.TestBase {
 construct() {
 ...
 }
 ...
}
```

#### To run a GUnit test

1. In the Project window, navigate to configuration → gtest, and then to your test class.
2. Right-click the test, and then click either Run ‘TestName’ or Debug ‘TestName’. This action opens a test console at the bottom of the screen.
3. (Optional) If desired, you can also create individual run/debug settings to use while running this test class. For details, see “Configuring the Test Environment” on page 426.

## Using Entity Builders to Create Test Data

**Note:** Guidewire does not recommend or support the use of classes that extend `gw.api.databuilder.DataBuilder` or classes that reside in the `gw.api.databuilder.*` package in a production environment. Guidewire provides GUnit as a development test facility only.

As you run tests against code, you need to run these test in the context of a known set of data objects. This set of objects is generally known as a *test fixture*. You use Gosu entity builders to create the set of data objects to use in testing.

Guidewire provides a number of entity “builders” as utility classes to quickly and concisely create objects (entities) to use as test data. The PolicyCenter base configuration provides builders for the base entities (like PolicyBuilder, for example). However, if desired, you can extend the base `DataBuilder` class to create new or extended entities. You can commit any test data that you create using builders to the test database using the `bundle.commit` method.

For example, the following builder creates a new `Person` object with a `FirstName` property set to “Sean” and a `LastName` property set to “Daniels”. It also adds the new object to the default test bundle.

```
var myPerson = new PersonBuilder()
 .withFirstName("Sean")
 .withLastName("Daniels")
 .create()
```

For readability, Guidewire recommends that you place each configuration method call on an indented separate line starting with the dot. This makes code completion easier. It also makes it simpler to alter a line or paste a new line into the middle of the chain or to comment out a line.

Gosu builders extend from the base class `gw.api.databuilder.DataBuilder`. To view a list of valid builder types in Guidewire PolicyCenter, use the Studio code completion feature. Enter `gw.api.databuilder.` in the Gosu editor and Studio displays the list of available builders.

### Package Completion

As you create an entity builder, you must either use the full package path, or add a `uses` statement at the beginning of the test file. However, in general, Guidewire recommends that you place the package path in a `uses` statement at the beginning of the file.

```
uses gw.api.builder.AccountBuilder

@gw.testharness.ServerTest
class MyTest extends TestBase {

 construct(testname : String) {
 super(testname)
 }
 ...
 function testSomething() {
 //perform some test
 var account = new AccountBuilder().create()
 }
 ...
}
```

Or, more simply (although Guidewire does not recommend this), enter the full path within the test class itself:

```
var account = new gw.api.builder.AccountBuilder().create()
```

Guidewire provides certain of the `Builder` classes in `gw.api.builder.*` and others in `gw.api.databuilder`. Verify the package path as you create new builders.

## Creating an Entity Builder

To create a new entity builder of a particular type, you merely need to use the following syntax:

```
new TypeOfBuilder()
```

This creates a new builder of the specified type, with the `Builder` class setting various default properties on the builder entity. (Each entity builder provides different default property values depending on its particular implementation.) For example, to create (or build) a default address, use the following:

```
var address = new AddressBuilder()
```

To set specific properties to specific values, you need to also use the property configuration methods. There are three different types of property configuration methods, each which serves a different purpose as indicated by the method's initial word.

Initial word	Indicates
on	A link to a parent, for example, <code>PolicyPeriod</code> is on an <code>Account</code> , so the method is <code>onAccount(Account account)</code> .
as	A property that holds only a single state, for example, <code>asBusinessType</code> or <code>asAgencyBill</code> .
with	The single element or property to be set. For example, the following sets a <code>FirstName</code> property: <code>withFirstName("Joe")</code>

Use a `DataBuilder.with(...)` configuration method to add a single property or value to a builder object. For example, the following Gosu code creates a new `Address` object and uses a number of `with(...)` methods to initialize properties on the new object. It then uses an `asType(...)` method to set the address type.

```
var address = new AddressBuilder()
 .withAddressLine1(codeStr1 + " Main St.")
 .withAddressLine2("Suite " + codeStr2)
 .withCity("San Mateo")
 .withState("CA")
 .withPostalCode("94404-" + codeStr3)
```

```
.asBusinessType()
...
```

After you create a builder entity, you are responsible for writing that entity to the database as part of a transaction bundle. In most cases, you must use one of the builder `create` methods to add the entity to a bundle. Which `create` method one you choose depends on your purpose.

To complete the previous example, you need to add a `create` method at the end.

```
var address = new AddressBuilder()
 .withAddressLine1(codeStr + " Main St.")
 ...
 .create()
```

## Builder Create Methods

The `DataBuilder` class provides the following `create` methods:

```
builderObject.create(bundle)
builderObject.create()
builderObject.createAndCommit()
```

The following list describes these `create` methods.

Method	Description
<code>create()</code>	Creates an instance of this builder's entity type, in the default bundle. This method does not commit the bundle. Studio resets the default bundle before every test class and method.
<code>createAndCommit()</code>	Creates an instance of this builder's entity type, in the default bundle and performs a commit of that default bundle.
<code>create(bundle)</code>	Creates an instance of this builder's entity type, with values determined by prior calls to the entity. The bundle parameter sets the bundle to use while creating this builder instance.

### The No-Argument Create Method

The no-argument `create` method uses a default bundle that all the builders share. This is adequate for most test purposes. However, as all objects created this way share the same bundle, committing the bundle on just one of the created objects commits all of the objects to the database. This also makes them available to the PolicyCenter interface portion of a test. For example:

```
var address = new AddressBuilder()
 .withCity("Springfield")
 .asHomeAddress()
 .create()

new PersonBuilder()
 .withFirstName("Sean")
 .withLastName("Daniels")
 .withPrimaryAddress(address)
 .create()

address.Bundle.commit()
```

In this example, `Address` and `Person` share a bundle, so committing `address.Bundle` also stores `Person` in the database. If you do not need a reference to the `Person`, then you do not need to store it into a variable.

JUnit resets the default bundle before every test class and method.

### The Create and Commit Method

The `createAndCommit` method is similar to the `create` method in that it adds the entity to the default bundle. It then, however, commits that bundle to the database.

### The Create with Bundle Method

If you need to work with a specific bundle, use the `create(bundle)` method. Guidewire recommends that you use this method inside of a transaction block. A transaction block provides the following:

- It creates the bundle at the same time as it creates the new builder.

- It automatically commits the bundle as it exits.

The following example illustrates the use of a data builder inside a transaction block.

```
function myTest() {
 var person : Person

 Transaction.RunWithNewBundle(\ bundle -> {
 person = new PersonBuilder()
 .withFirstName("John")
 .withLastName("Doe")
 .withPrimaryAddress(new AddressBuilder()
 .withCity("Springfield")
 .asHomeAddress())
 .create(bundle)
 })

 assertEquals("Doe", person.LastName)
}
```

Notice the following about this example:

- The example declares the `person` variable outside the transaction block, making it accessible elsewhere in the method.
- The data builder uses an `AddressBuilder` object nested inside `PersonBuilder` to build the address.
- The `Transaction.RunWithNewBundle` statement creates the bundle and automatically commits it after Gosu Runtime executes the supplied code block.

In summary, the `create(bundle)` method does not create a bundle. Rather, it uses the bundle passed into it. Guidewire recommends that you use this method inside a transaction block that both creates the bundle and commits it automatically.

If you do not use this method inside a transaction block that automatically commits a bundle, then you must commit the bundle yourself. To do so, add `bundle.commit` to your code.

## Entity Builder Examples

The following examples illustrate various ways that you can use builders to create sample data for use in GUnit tests.

- Creating Multiple Objects from a Single Builder
- Nesting Builders
- Overriding Default Builder Properties

### Creating Multiple Objects from a Single Builder

The Builder class creates the builder object at the time of the `create` call. Therefore, you can use the same builder instance to generate multiple objects.

```
var activity1 : Activity
var activity2 : Activity
var bundle = gw.transaction.Transaction.RunWithNewBundle(\ bundle -> {
 var activityBuilder = new gw.api.builder.ActivityBuilder()
 .withType("general")
 .withPriority("high")
 activity1 = activityBuilder.withSubject("this is test activity one").create(bundle)
 activity2 = activityBuilder.withSubject("this is test activity two").create(bundle)
})
```

## Nesting Builders

It is possible to nest one builder inside of another by having a method on a builder that takes another builder as an argument. For example, suppose that you want to create an Account that has a Policy. In this situation, you might want to do the following:

```
Account account = new AccountBuilder()
 .withPolicies(new PolicyBuilder().withDefaultPolicyPeriod())
 .create()
```

## Overriding Default Builder Properties

The following code samples illustrates multiple ways to create an Account object. The first code sample shows a simple test method and uses a transaction block. The `Transaction` object takes a block, which assigns the new account to the variable in the scope outside of the transaction.

```
function myTest(){
 var account : Account
 Transaction.runWithNewBundle(\ bundle -> {
 account = new AccountBuilder().create(bundle)
 })
}
```

There are generally two kinds of accounts: person and company. By default, `AccountBuilder` creates a person account. If you want a company account, then you need to assign a company contact as the account holder, as shown in the following code sample:

```
account = new AccountBuilder(false)
 .withAccountHolderContact(new PolicyCompanyBuilder(42))
 .create(bundle)
}
```

In this example, passing `false` to `AccountBuilder` tells it not to create a default account holder. Instead, you pass in your own account holder by calling `withAccountHolderContact`, which takes a `ContactBuilder`. In this case, `PolicyCompanyBuilder` suffices. The passed in number 42 seeds the default data with something unique (ideally) and identifiable.

The following example creates a company account and overrides some of the default values. Anywhere you see `code`, it means numerical seed value. (String variants derive from the given values.) It also illustrates how to nest the results of one builder inside another.

```
var address = new AddressBuilder()
 .withAddressLine1(codeStr + " Main St.")
 .withAddressLine2("Suite " + codeStr)
 .withCity("San Mateo")
 .withState("CA")
 .withPostalCode("94404-" + codeStr)
 .asBusinessType()

var company = new PolicyCompanyBuilder(code, false)
 .withCompanyName("This Company " + code)
 .withWorkPhone("650-555-" + codeStr)
 .withAddress(address)
 .withOfficialID(new OfficialIDBuilder().withType("FEIN").withValue("11-222" + codeStr))

var account = new AccountBuilder(false)
 .withIndustryCode("1011", "SIC")
 .withAccountOrgType("Corporation")
 .withAccountHolderContact(company)
 .create(bundle)
```

The following example takes the previous code and presents it as a single builder that takes other builders as arguments. While more compact, it also takes more planning and understanding of builders to create. Notice the successive levels of indenting used to signal the creation of a new (embedded) builder.

```
var account = new AccountBuilder(false)
 .withIndustryCode("1011", "SIC")
 .withAccountOrgType("Corporation")
 .withAccountHolderContact(new PolicyCompanyBuilder(code, false)
 .withCompanyName("This Company " + code)
 .withWorkPhone("650-555-" + codeStr)
 .withAddress(new AddressBuilder()
 .withAddressLine1(codeStr + " Main St.")
 .withAddressLine2("Suite " + codeStr)
```

```
 .withCity("San Mateo")
 .withState("CA")
 .withPostalCode("94404-" + codeStr)
 .asBusinessType()
 .withOfficialID(new OfficialIDBuilder()
 .withType("FEIN")
 .WithValue("11-222" + codeStr))
)
.create(bundle)
```

## Creating New Builders

If you need additional builder functionality than that provided by the PolicyCenter base configuration builders, you can do either of the following:

- Extend an existing builder class and add new builder methods to that class.
- Extend the base `DataBuilder` class and create a new builder class with its own set of builder methods.

You can also create a builder (by extending the `DataBuilder` class) for a custom entity that you created, if desired.

For more information, see the following:

- “Extending an Existing Builder Class” on page 435
- “Extending the `DataBuilder` Class” on page 436
- “Creating a Builder for a Custom Entity and Testing It” on page 437

## Extending an Existing Builder Class

To extend an existing builder class, use the following syntax:

```
class MyExtendedBuilder extends SomeExistingBuilder {
 construct() {
 ...
 }
 ...
 function someNewFunction() : MyExtendedBuilder {
 ...
 return this
 }
 ...
}
```

The following `MyPersonBuilder` class extends the existing `PersonBuilder` class. The existing `PersonBuilder` class contains methods to set both the first and last names of the person, but not the person’s middle name. The new extended class contains a single method to set the person’s middle name. As there is no static field for the properties on a type, you must look up the property by name.

```
uses gw.api.databuilder.PersonBuilder

class MyPersonBuilder extends PersonBuilder {

 construct() {
 super(true)
 }

 function withMiddleName(testname : String) : MyPersonBuilder {
 set(Person.TypeInfo.getProperty("MiddleName"), testname)
 return this
 }
}
```

The `PersonBuilder` class has two constructors. This code sample uses the one that takes a Boolean that means create this class `withDefaultOfficialID`.

Another more slightly complex example would be if you extended the `Person` object and added a new `PreferredName` property. In this case, you might want to extend the `PersonBuilder` class also and add a `withPreferredName` method to populate that field through a builder.

## Extending the DataBuilder Class

To extend the `DataBuilder` class, use the following syntax:

```
class MyNewBuilder extends DataBuilder<BuilderEntity, BuilderType> {
 ...
}
```

The `DataBuilder` class takes the following parameters:

Parameter	Description
<code>BuilderEntity</code>	Type of entity created by the builder. The <code>create</code> method requires this parameter so that it can return a strongly-typed value and, so that other builder methods can declare strongly-typed parameters.
<code>BuilderType</code>	Type of the builder itself. The <code>with</code> methods require this on the <code>DataBuilder</code> class so that it can return a strongly-typed builder value (to facilitate the chaining of <code>with</code> methods).

If you choose to extend the `DataBuilder` class (`gw.api.databuilder.DataBuilder`), place your newly created builder class in the `gw.api.databuilder` package in the Studio Tests folder. Start any method that you define in your new builder with one of the recommended words (described previously in “Creating an Entity Builder” on page 431):

Initial word	Indicates
<code>on</code>	A link to a parent, for example, <code>PolicyPeriod</code> is on an <code>Account</code> , so the method is <code>onAccount(Account account)</code> .
<code>as</code>	A property that holds only a single state, for example: <code>asBusinessType</code> or <code>as AgencyBill</code> .
<code>with</code>	The single element or property to be set. For example, the following sets a <code>FirstName</code> property: <code>withFirstName("Joe")</code>

Your configuration methods can set properties by calling `DataBuilder.set` and `DataBuilder.addArrayElement`. You can provide property values as any of the following:

- Simple values.
- Beans to be used as subobjects.
- Other builders, which PolicyCenter uses to create subobjects if it calls your builder's `create` method.
- Instances of `gw.api.databuilder.ValueGenerator`, which can, for example, generate a different value (to satisfy uniqueness constraints) for each instance constructed.

`DataBuilder.set` and `DataBuilder.addArrayElement` optionally accept an integer order argument that determines how PolicyCenter configures that property on the target object. (PolicyCenter processes properties in ascending order.) If you do not provide an order for a property, Studio uses `DataBuilder.DEFAULT_ORDER` as the order for that property. PolicyCenter processes properties with the same order value (for example, all those that do not have an order) in the order in which they are set on the builder.

In most cases, Guidewire recommends that you omit the order value as you are implementing builder configuration methods. This enables callers of your builder to select the execution order through the order of the configuration method calls.

Constructors for builders can call `set`, and similar methods to set up default values. These are useful to satisfy `null` constraints so it is possible to commit built objects to the database. However, Guidewire generally recommends that you limit the number of defaults. This is so that you have the maximum control over the target object.

## Other DataBuilder Classes

The `gw.api.databuilder` package also includes `gw.api.databuilder.ValueGenerator`. You can use this class, for example, to generate a different value for each instance constructed to satisfy uniqueness constraints. The `databuilder` package includes `ValueGenerator` class variants for generating unique integers, strings, and type-keys:

- `gw.api.databuilder.IntegerStringGenerator`
- `gw.api.databuilder.SequentialStringGenerator`
- `gw.api.databuilderTypekeyStringGenerator`

## Custom Builder Populators

Ideally, all building can be done through simple property setters, using the `DataBuilder.set` or `DataBuilder.addArrayElements` methods. However, you may want to define more complex logic, if these methods do not suffice. To achieve this, you can define a custom implementation of `gw.api.databuilder.populator.BeanPopulator` and pass it to `DataBuilder.addPopulator`. Guidewire provides an abstract implementation, `AbstractBeanPopulator`, to support short anonymous `BeanPopulator` objects.

The following example uses an anonymous subclass of `AbstractBeanPopulator` to call the `withCustomSetting` method. This code passes the group to the constructor, and the code inside of `execute` only accesses it through the `vals` argument. This allows the super-class to handle packaging details.

```
public MyEntityBuilder withCustomSetting(group : Group) {
 addPopulator(new AbstractBeanPopulator<MyEntity>(group) {
 function execute(e : MyEntity, vals : Object[]) {
 e.customGroupSet(vals[0] as Group)
 }
 }
 return this
}
```

The `AbstractBeanPopulator` class automatically converts builders to beans. That is, if you pass a builder to the constructor of `AbstractBeanPopulator`, it returns the bean that it builds in the `execute` method. The following example illustrates this.

```
public MyEntityBuilder withCustomSetting(groupBuilder : DataBuilder<Group, ?>) : MyEntityBuilder {
 addPopulator(new AbstractBeanPopulator<MyEntity>(groupBuilder) {
 function execute(e : MyEntity, vals : Object[]) {
 e.customGroupSet(vals[0] as Group)
 }
 }
 return this
}
```

## Creating a Builder for a Custom Entity and Testing It

It is also possible, if you want, to create a builder for a custom entity. For example, suppose that you want each PolicyCenter user to have an array of external credentials (for automatic sign-on to linked external systems, perhaps). To implement, you can create an array of `ExtCredential` on `User`, with each `ExtCredential` having the following parameters:

Parameter	Type
ExtSystem	Typekey
UserName	String
Password	String

After creating your custom entity and its builder class, you would probably want to test it. To accomplish this, you need to do the following:

Task	Affected files	See
1. Create a custom ExtCredential array entity and extend the User entity to include it.	ExtCredential.eti User.etx	To create a custom entity
2. Create an ExtCredentialBuilder by extending the DataBuilder class and adding withXXX methods to it.	ExtCredentialBuilder.gs	To create an ExtCredentialBuilder class
3. Create a test class to exercise and test your new builder.	ExtCredentialBuilderTest.gs	To create an ExtCredentialBuilderTest class

### To create a custom entity

To create a new array ExtCredential custom entity, you need to do the following:

- Add the ExtSystem typelist (in the **Typelist** editor in Guidewire Studio).
- Define the ExtCredential array entity (in ExtCredential.eti, accessible through Guidewire Studio).
- Modify the array entity definition to include a foreign key to User (in ExtCredential.eti).
- Add an array field to the User entity (in User.etx).

**1.** Add an ExtSystem typelist. Within Guidewire Studio, navigate to **Typelist**, and then right-click **New → Typelist**. Add a few *external system* typecodes. (For example, add SystemOne, SystemTwo, or similar items.)

**2.** Create ExtCredential1. Right-click **Entity**, and then click **New → Entity**. Name this file ExtCredential.eti and enter the following:

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel" entity="ExtCredential" table="extcred"
 type="retireable" exportable="true" platerform="true" >
 <typekey name="ExtSystem" typelist="ExtSystemType" desc="Type of external system"/>
 <column name="UserName" type="shorttext"/>
 <column name="Password" type="shorttext"/>
 <foreignkey name="UserID" fkentity="User" desc="FK back to User"/>
</entity>
```

**3.** Modify the User entity. Find User.etx (in **Extensions → Entity**). If it does not exist, then you must create it.

However, most likely, this file exists. Open the file and add the following:

```
<array name="ExtCredentialRetirable" arrayentity="ExtCredential"
 desc="An array of ExtCredential objects" arrayfield="UserID" exportable="false"/>
```

See “Extending a Base Configuration Entity” on page 215 for information on extending the Guidewire PolicyCenter base configuration entities.

### To create an ExtCredentialBuilder class

Next, you need to extend the base DataBuilder class to create the ExtCredentialBuilder class. Place this class in its own package in the **Classes** folder.

For example:

```
package AllMyClasses

uses gw.api.databuilder.DataBuilder

class ExtCredentialBuilder extends DataBuilder<ExtCredential, ExtCredentialBuilder> {

 construct() {
 super(ExtCredential)
 }

 function withType (type: typekey.ExtSystemType) : ExtCredentialBuilder {
 set(ExtCredential.TypeInfo.getProperty("ExtSystem"), type)
 return this
 }
}
```

```

function withUserName(somename : String) : ExtCredentialBuilder {
 set(ExtCredential.TypeInfo.getProperty("UserName"), somename)
 return this
}

function withPassword(password : String) : ExtCredentialBuilder {
 set(ExtCredential.TypeInfo.getProperty("Password"), password)
 return this
}

}

```

Notice the following about this code sample:

- It includes a `uses ... DataBuilder` statement.
- It extends the `DataBuilder` class, setting the `BuilderType` parameter to `ExtCredential` and the `BuilderEntity` parameter to `ExtCredentialBuilder`. (See “Extending the DataBuilder Class” on page 436 for a discussion of these two parameters.)
- It uses a constructor for the super class—`DataBuilder`—that requires the entity type to create.
- It implements multiple `withXXX` methods that populate an `ExtCredential` array object with the passed in values.

#### To create an `ExtCredentialBuilderTest` class

Finally, to be useful, you need to reference your new builder in Gosu code. You can, for example, create a GUnit test that uses the `ExtCredentialBuilder` class to create test data. Place this class in its own package in the `Tests` folder.

```

package MyTests

uses AllMyClasses.ExtCredentialBuilder
uses gw.transaction.Transaction

@gw.testharness.ServerTest
class ExtCredentialBuilderTest extends gw.testharness.TestBase {

 static var credential : ExtCredential
 construct() {

 }

 function beforeClass () {
 Transaction.runWithNewBundle(\ bundle -> {
 credential = new ExtCredentialBuilder()
 .withType("SystemOne")
 .withUserName("Peter Rabbit")
 .withPassword("carrots")
 .create(bundle)
 })
 }

 function testUsername() {
 assertEquals("User names do not match.", credential.UserName, "Peter Rabbit")
 }

 function testPassword() {
 assertEquals("Passwords do not match.", credential.Password, "carrots")
 }
}

```

Notice the following about this code sample:

- It includes the `uses` statements for both `ExtCredentialBuilder` and `gw.transaction.Transaction`.
- It creates a static `credential` variable. As the code declares this variable outside of a method—as a class variable—it is available to all methods within the class. (GUnit maintains a single copy of this variable.) As you run a test, GUnit creates a single instance of the test class that each test method uses. Therefore, to preserve a variable value across multiple test methods, you must declare it as a static variable. (For a description of the `static` keyword and how to use it in Gosu, see “Static Modifier” on page 206 in the *Gosu Reference Guide*.)

- It uses a `beforeClass` method to create the `ExtCredential` test data. This method calls `ExtCredentialBuilder` as part of a transaction block, which creates and commits the bundle automatically. GUnit calls the `beforeClass` method before it instantiates the test class for the first time. Thereafter, the test class uses the test data created by the `beforeClass` method. It is important to understand that GUnit does not drop the database between execution of each test method within a test class. However, if you run multiple test classes together (for example, by running all the test classes in a package), GUnit resets the database between execution of each test class.
- It defines several test methods, each of which starts with `test`, with each method including an `assertXXX` method to test the data.

If you run the `ExtCredentialBuilderTest` class as defined, the GUnit tester displays green icons, indicating that the tests were successful:

---



part VIII

# Guidewire PolicyCenter Configuration



# PolicyCenter Configuration Guidelines

This topic provides guidelines for configuring PolicyCenter.

This topic includes:

- “Guidelines for Modularizing Line-of-business Code” on page 443

## Guidelines for Modularizing Line-of-business Code

Guidewire recommends that you define line-of-business code in the policy-line-methods classes. Avoid putting line-of-business code in generic locations such as the rule sets, plugins, and non-line-of-business PCF files and Gosu classes. This recommendation is intended to make upgrade easier by grouping line-of-business changes in the `gw.lob` package.

### Example

The policy period plugin implementation (`gw.plugin.policyperiod.impl.PolicyPeriodPlugin`) implements the `postApplyChangesFromBranch` method. Rather than putting code for each line of business directly in the plugin implementation, 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) {
 ...
}
```



# Configuring Policy Archiving

*Archiving* is the process of moving the data associated with a policy from the active PolicyCenter database to a document storage area. You can still search for and retrieve archived policies. But, while archived, these policies occupy less space in the active database.

*This topic includes:*

- “Archiving and the Domain Graph” on page 445
- “Archiving in Guidewire PolicyCenter” on page 447
- “Archiving and Encryption” on page 448
- “Selecting Policy Terms for Archive Eligibility” on page 448
- “Batch Processes and Archiving” on page 449
- “Monitoring Archiving Activity” on page 450
- “Configuring Archiving” on page 450
- “Archiving Plugins” on page 454
- “Archiving Test Tool” on page 455

**See also**

- “More Information on Archiving” on page 325 in the *Application Guide* for a list of topics related to archiving.

---

**IMPORTANT** Guidewire strongly recommends that you contact Customer Support before implementing archiving.

---

## Archiving and the Domain Graph

Guidewire PolicyCenter uses the domain graph to define the aggregate cluster of associated objects that it treats as a single unit for purposes of archiving. Each aggregate cluster has a root and a boundary.

- The *root* is a single specific entity that the aggregate cluster contains. The root entity is the main entity in the graph. A root entity is application-specific.

In PolicyCenter, the root entity is the `PolicyPeriod`. During the archiving of an instance of the domain graph, PolicyCenter leaves behind the `PolicyPeriod`, its related `EffectiveDatedFields`, and optionally other objects. For more information about the objects that remain in PolicyCenter, see “Entities Retained After Archiving” on page 321 in the *Application Guide*.

- The *boundary* defines what is inside the aggregate cluster of objects. Or, in other words, it identifies all of the entities that are part of the graph.

In PolicyCenter, the boundary defines the entities that relate to a `PolicyPeriod`, such as jobs and effective dated fields.

A domain graph defines the unit of work for object archiving. The unit of work for the archive process is a single instance of the domain graph. For example, a unit of work is a single policy period and all its associated entities. While the archiving process works on individual policy period graphs, higher level code considers the policy term as the logical unit of archiving for PolicyCenter.

To enforce the boundaries of the domain graph, all objects participating in the archive process must implement one or more of the following delegates:

Delegate	Reason for use...
<code>RootInfo</code>	<p>During the archiving of an instance of the domain graph, PolicyCenter keeps the <code>PolicyPeriod</code>, its related <code>EffectiveDatedFields</code>, and optionally other objects in the main PolicyCenter database. For more information about the objects that remain in PolicyCenter, see “Entities Retained After Archiving” on page 321 in the <i>Application Guide</i>.</p> <p>The <code>PolicyPeriod</code> instance provides the following:</p> <ul style="list-style-type: none"> <li>Sufficient information to retrieve the data</li> <li>Sufficient information for a minimal search on archived data</li> </ul> <p>The policy period entity—and only this entity—implements the <code>RootInfo</code> delegate. The policy period entity is the root object in the PolicyCenter domain graph. You cannot change which entity implements the <code>RootInfo</code> delegate.</p>
<code>Extractable</code>	<p>All entities in the domain graph must implement the <code>Extractable</code> delegate. The converse is also true. No entity outside the domain graph can implement the <code>Extractable</code> delegate.</p> <p>The use of the <code>Extractable</code> delegate ensures the creation of the <code>ArchivePartition</code> column that PolicyCenter uses during the archive process.</p>
<code>OverlapTable</code>	<p>Overlap tables are tables in which each individual table row is either in the domain graph or outside of it (part of reference data), but not both. Entity types corresponding to overlap tables must implement the <code>OverlapTable</code> delegate. Implementing the <code>OverlapTable</code> delegate creates an additional <code>Admin</code> column that PolicyCenter uses to determine which individual rows belong to the domain graph, and which do not. As these objects are both inside and outside the domain graph, they must also implement the <code>Extractable</code> delegate.</p>

**Data model delegate objects.** A `Delegate` is a reusable type that defines database columns, an interface, and a default implementation of that interface. A delegate permits an entity to implement an interface while delegating the implementation of that interface to another class, that of the delegate. Each delegate type provides additional columns on the affected tables.

#### See also

- For a discussion of object related to archiving, see “Data Model for Archiving” on page 450.
- For a discussion of the Guidewire data model in general, see “The PolicyCenter Data Model” on page 149.
- For a discussion of delegate objects and how to work with them, see “Delegate Data Objects” on page 161.
- For a discussion of the domain graph, see “The Archiving Domain Graph” on page 247.

## The Root Info Entity

The `PolicyPeriod` entity is the root info entity instance that remains in the active database after PolicyCenter archives the policy term. Searches occur only in the active database and can find both active and archived policies. Search can only find archived policies if the search is based on fields retained in the active database.

A `PolicyPeriod` instance retains some links to entities which are outside the domain graph and remain in the PolicyCenter database. For examples, see `Policy` and `PolicyTerm`.

Because it implements the `RootInfo` delegate, the `PolicyPeriod` entity includes the `ArchiveState` column, which records the archive state. The `ArchiveState` value, from the `ArchiveState` typelist, is one of the following:

- `archived`
- `retrieving`
- `null`

If `ArchiveState` is `null`, the policy is in the active PolicyCenter database, either because the policy has never been archived or because it was successfully retrieved from the archive.

The logical unit of archiving is the policy term. Therefore, application and user interface code can check the `PolicyTerm.Archived` flag to determine if a given entity is archived or not.

## Archiving Objects Example

When you archive a policy, PolicyCenter archives the `PolicyTerm`. When the policy term is archived, internal code archives each `PolicyPeriod` in that term. In the usual case, all policy periods in that term are archived. However, an error can occur and one or more policy periods may not successfully complete archiving.

PolicyCenter sets the `Archived` bit to true on `PolicyTerm` if at least one `PolicyPeriod` has the `Archived` bit set to true. After archiving, the `PolicyTerm` and `PolicyPeriod` entities remain in the PolicyCenter database.

## Archiving in Guidewire PolicyCenter

Archiving is a multi-step process. At a high level, these steps involve the following:

1. The Archive Policy Terms batch process queries the database to select policy terms that are potentially eligible for archiving. The batch process creates a `WorkItem` in the database for each eligible policy term.
2. Individual archive workers pick up—one at a time—any archiving work items created during the first step. They then run further checks on each identified policy term and the policy periods in that term. If the checks do not pass, the writer skips that policy term. PolicyCenter marks the date on which to check it again in a configuration parameter.  
These eligibility checks are part of the underlying archiving architecture and are not configurable. For example, internal checks prevent PolicyCenter from archiving a policy term with a policy period that is part of an unfinished workflow.  
Guidewire provides other eligibility checks you can configure, enable or disable, and modify in the `IPCArchivingPlugin`.
3. Finally, if all checks pass, a worker moves the domain graph data from the PolicyCenter database to the archive backing store. PolicyCenter manages the movement of data from the database to the archive backing store through the use of the `IArchiveSource` plugin. Guidewire provides the implementation of this plugin in the base configuration as an example only. *You must implement your own production archive source plugin.*  
Although PolicyCenter archives the policy term, the worker archives the individual policy periods, one at a time in an order defined by internal code. The archiving of each policy period is a database transaction.

**See also**

- “Archiving Plugins” on page 454 for a brief description of the archiving plugins.
- “Archiving Integration” on page 555 in the *Integration Guide* for detailed information about the archiving plugins.
- “Selecting Policy Terms for Archive Eligibility” on page 448

## Archiving and Encryption

You are responsible for implementing the `IArchiveSource` plugin interface in such a way as to provide any necessary data encryption. PolicyCenter does not encrypt the values in the XML that it generates and passes to the `IArchiveSource` plugin. However, PolicyCenter does provide information about which properties are marked as encrypted in the data model in the XSD.

You must implement encryption in the `IArchiveSource` plugin if you want it. You must also decide what to encrypt. In many cases, the entire document is encrypted. It is up to you to determine the scheme to use for managing the encryption keys.

## Selecting Policy Terms for Archive Eligibility

PolicyCenter calls the `getArchiveDate` method of the `IPCArchivingPlugin` to determine the date on which a policy term is eligible for archiving. In the default configuration, the `PCArchivingPlugin` class implements this plugin. In `PCArchivingPlugin`, `getArchiveDate` determines the date for archive eligibility as follows:

- If any open claims exist, delay `ArchiveDefaultRecheckDays` from today.
- Any open jobs, open activities, or active messages result in a delay of `ArchiveDefaultRecheckDays` days from today.
- Any scheduled audits result in a delay of `ArchiveRecentJobCompletionDays` days from initialization date of the last scheduled audit.
- Any active jobs based on policy periods from the policy term result in a delay of `ArchiveDefaultRecheckDays` days from today.
- Any recently completed jobs result in a delay of `ArchiveRecentJobCompletionDays` days from the close date of the last completed job.

Both `ArchiveDefaultRecheckDays` and `ArchiveRecentJobCompletionDays` are configuration parameters.

If none of the above criteria apply or the dates calculated are prior to today, the archive date is the current date. Otherwise, the archive date is the maximum date from the bullet items.

A policy can be excluded from archiving. For more information, see “Do Not Archive a Policy” on page 453.

## Configuring Archive Eligibility

Implement the `IPCArchivingPlugin` interface to configure whether open claims delay archiving. You can also configure delays to archiving for other reasons. In the default configuration, `PCArchivingPlugin` implements this interface.

The `getArchiveDate` method of `IPCArchivingPlugin` is called by the `calculateNextArchiveCheckDate` method of `PCArchivingUtil`.

**See also**

- “Plugin to Determine Archive Eligibility” on page 454

## Batch Processes and Archiving

PolicyCenter has batch processes that archive policy terms and retrieve policy terms.

Name	Code	Description
Archive policy terms	ArchivePolicyTerm	<p>Run this batch process to archive policy terms. This batch process calls the <code>IPCArchivingPlugin</code> which determines whether a policy can be archived.</p> <p>For a policy period to be eligible for archiving, the server time must have reached the <code>PolicyTerm.NextArchiveCheckDate</code> date:</p> <ul style="list-style-type: none"><li>• For more information on policy term eligibility, see “Selecting Policy Terms for Archive Eligibility” on page 448.</li><li>• For information on the archive work queue, and those policy periods that the archive process archived, excluded, or skipped, see “Archive Info” on page 144 in the <i>System Administration Guide</i>.</li></ul> <p>After running this work queue, Guidewire recommends that you update database statistics. This work queue makes large changes to the tables. Updating database statistics enables the optimizer to pick better queries based on more current data. For instructions to gather database statistics, see “Configuring Database Statistics” on page 42 in the <i>System Administration Guide</i>.</p> <p>For more information, see “Archiving Policy Terms” on page 320 in the <i>Application Guide</i> and “Archiving-related Configuration Parameters” on page 452.</p>
Retrieve policy terms	RestorePolicyTerm	Run this batch process to retrieve archived policy terms that are marked for retrieval. You can request retrieval of an archived policy term. Retrieving a policy term generates an activity for the user who requested the retrieval. For more information, see “Retrieving Archived Policies” on page 324 in the <i>Application Guide</i> .

**Note:** PolicyCenter does not use the `Archiving Item` writer work queue.

### See also

- “Scheduling Work Queues and Batch Processes” on page 107 in the *System Administration Guide*
- “Retrieving Archived Policies” on page 324 in the *Application Guide*

## Batch Processes to Run Prior to Archive Policy Term Batch Process

PolicyCenter will not archive a policy term if it is associated with one or more workflows. To more effectively archive policy terms, clean up workflows before archiving by running the following batch processes:

- Purge Workflow
- Purge Workflow Logs
- Workflow

## Monitoring Archiving Activity

PolicyCenter provides server tools to help you monitor and supervise the archiving process:

Tool	Description
Work Queue Info	The Server Tools → Work Queue Info page shows the status of the archive work queue. You can use tools on this page to run a work queue writer and to stop and restart workers.
Archive Info	<p>The Server Tools → Info Pages → Archive Info page provides status information about the archive process. It includes information on the following:</p> <ul style="list-style-type: none"><li>• Entities archived</li><li>• Entities excluded because of business logic</li><li>• Entities excluded because of failure</li></ul> <p>The Archive Info page provides tools to reset various archive items as well. See “Info Pages” on page 144 in the <i>System Administration Guide</i> for more information.</p>

### Errors in the Archiving Process

If an API or user interface operation attempts to open or work on an archived policy term or policy period, PolicyCenter typically generates an `PolicyTermInArchiveException`. If an archive worker cannot archive a policy period for any reason, it flags the policy term as `ExcludedFromArchive`. The Archive Info page `Excluded from Archive` shows the number of excluded policy periods.

### Viewing Archiving Log Activity

It is possible to create a separate log for successfully archived objects. The log contains a list of the objects that PolicyCenter successfully archived.

To configure these log messages, uncomment and, if necessary, edit the archive success logger in the `logging.properties` file. This logger is:

```
log4j.category.Server.Archiving.Success
```

For information about logging and how to modify `logging.properties`, see “Logging Successfully Archived Policy Terms and Policy Periods” on page 25 in the *System Administration Guide*.

## Configuring Archiving

In working with archiving, you can configure the following:

- Data Model for Archiving
- Archiving-related Configuration Parameters
- Gosu Code and Archiving
- Archive Work Queue

### Data Model for Archiving

This topic describes the data model for archiving.

## Policy

The Policy entity has the following properties related to archiving:

Property	Description
DoNotArchive	Do not archive any of the terms for this policy. If you change this property from true to false, terms already archived are not automatically retrieved.

## PolicyTerm

The PolicyTerm entity has the following properties related to archiving:

Property	Description
Archived	Value is true if at least one policy period associated with this policy term is archived.
NextArchiveCheckDate	The date to next evaluate this PolicyTerm for archiving. Value is null if archiving is to be checked at the next opportunity.
RestoreRequests	Array to PolicyTermRestoreRequest entity instances.

## PolicyTermRestoreRequest

The PolicyTermRestoreRequest entity represents a request made to retrieve this term from the archive and restore it to PolicyCenter. This entity has the following properties related to archiving:

Property	Description
PolicyTerm	A foreign key to the policy term to retrieve from the archive.
Reason	The reason provided by the user.
RequestingUser	The user that initiated this request to retrieve the policy term.
ShouldCreateActivity	A flag that indicates whether to create an activity after processing this request.

## PolicyPeriod

The PolicyPeriod entity has the following properties related to archiving:

Property	Description
Archived	This value is true if this policy period is archived.
ArchiveDate	The date on which archiving was attempted on the PolicyPeriod. Value is null if archiving has never been attempted.
ArchiveFailure	A foreign key to ArchiveFailure.
ArchiveFailureDetails	A foreign key to ArchiveFailureDetails.
ArchiveSchemaInfo	A foreign key to UpgradeDatamodelInfo, the schema version at which the PolicyPeriod was archived. Value is null if it was not archived.
ArchiveState	A type key that indicates the archive state of the graph. Values can be: <ul style="list-style-type: none"> <li>• archived – Graph has been archived</li> <li>• retrieving – Graph is marked for retrieving</li> </ul>
Locked	The row is locked and cannot be edited. Only a locked policy period can be archived.

## ArchiveFailure

The ArchiveFailure entity contains a short description of the reason that archiving failed.

## ArchiveFailureDetails

The ArchiveFailureDetails entity contains a detailed description of the reason that archiving failed.

## Archiving-related Configuration Parameters

Guidewire provides a set of configuration parameters that relate to the archive process. You use these parameters to enable and manage various aspects of the archive process. You set these configuration parameters in file config.xml.

Parameter	Type	Description
ArchiveEnabled	Boolean	<p><i>Required.</i> Whether archiving is enabled (set to true) or disabled (set to false). Default is false.</p> <p>This parameter controls the creation of indexes on the ArchivePartition column. If set to true, PolicyCenter creates a non-unique index on that column for Extractable entities.</p> <p>In PolicyCenter, this parameter controls whether or not the domain graph is started.</p> <p><b>WARNING</b> If you set ArchiveEnabled to true, the server refuses to start if you subsequently set it to false.</p> <p>See “ArchiveEnabled” on page 38.</p>
ArchiveDaysRetrievedBeforeArchive	Integer	<p>Minimum number of days that must pass after retrieving a term before it can be archived again.</p> <p>The default in the base configuration is 90.</p> <p>See “ArchiveDaysRetrievedBeforeArchive” on page 38.</p>
ArchiveDefaultRecheckDays	Integer	<p>The minimum number of days that must pass before a policy term can be reconsidered for archiving after a previous check did not identify it as an archiving candidate.</p> <p>The default in the base configuration is 30.</p> <p>See “ArchiveDefaultRecheckDays” on page 39.</p>
ArchivePolicyTermDays	Integer	<p>The minimum number of days that must pass before a given policy term can be archived. The minimum number of days begins after the PeriodEnd for policies that have at least one bound period and after the close date for withdrawn or not taken policies.</p> <p>The default in the base configuration is 366.</p> <p>See “ArchivePolicyTermDays” on page 39.</p>
ArchiveRecentJobCompletionDays	Integer	<p>The minimum number of days that must pass after a job is closed before archiving the associated policy term. Jobs are associated with policy periods.</p> <p>The default in the base configuration is 90.</p> <p>See “ArchiveRecentJobCompletionDays” on page 39.</p>

### See Also

- “Archive Parameters” on page 38
- “Domain Graph Parameters” on page 51

## Gosu Code and Archiving

After archiving, PolicyCenter deletes most of the `PolicyPeriod` subobjects, but keeps the `PolicyPeriod`, its associated `EffectiveDatedFields`, and optionally some other objects. The deleted subobjects are no longer accessible through Gosu code. For more information about the objects that remain in PolicyCenter, see “Entities Retained After Archiving” on page 321 in the *Application Guide*.

If archiving is enabled and your code accesses policy periods, your code needs to check whether that policy period is archived before attempting to traverse the subobjects. If your code attempts to traverse the subobjects in an archived `PolicyPeriod`, PolicyCenter throws an exception.

If there is a need to change an archived policy period, it must first be retrieved from the archive.

In the default configuration, PCF files and code checks for archiving before traversing a policy period. For example, you may see code checking for archiving in policy terms and policy periods such as:

- `aPolicyPeriod.PolicyTerm.Archived`
- `aPolicyPeriod.BasedOn.PolicyTerm.Archived` – If you are about to do a policy comparison, for example
- `aJob.SelectedVersion.PolicyTerm.Archived`

A `PolicyPeriod` has been archived if the `Archived` bit is set to `true`.

The `basedOn` property of a given `PolicyPeriod` might be in a prior term, so that based-on policy period must also be checked.

## Displaying Jobs or Policies

In the default configuration, the `JobForward` and `PolicyFileForward` PCF files are used by other PCF files and Gosu code to display a single job or policy. These PCF files figure out, among other things, whether or not the user arrives at the wizard for a job or policy or the archived policy period page instead.

A PCF file that displays or potentially modifies archivable entities calls this PCF file, which contains code to handle the display of an archived policy term or policy period. Care still needs to be taken with actions (such as calculating differences or starting jobs) whose success is dependent on other terms.

## Jobs and Archiving

When writing Gosu code, be aware that PolicyCenter has the following restriction related to jobs and archiving:

- You cannot start a job on an archived policy term.
- You cannot start a job if there is a future archived policy term. You cannot start a job because an out-of-sequence change may necessitate updates to future policy terms.
- You cannot archive a policy term if there are open jobs in a previous term.

## Search and Archiving

When writing Gosu code, be aware that search does not find properties on effective-dated objects removed from the database through archiving. The search simply fails to return those archived rows. Search screens may need to be aware of archiving and adjust the display when `Include Archived` or the like is selected.

## Do Not Archive a Policy

The `Policy` entity has a `DoNotArchive` property. You can manage this property through the `ArchiveAPI` web service. To set the `DoNotArchive` property, you must have the `archivedisable` and `archiveenable` privileges. In the default configuration, the user interface does not set or clear the `DoNotArchive` property.

### See also

- “Archiving Web Services” on page 119 in the *Integration Guide* for more information on the `IArchiveAPI`.

## Archive Work Queue

The Archive Policy Terms batch process writes items to the Archive work queue. Items in this queue correspond to possible archivable items. As PolicyCenter processes these items, it writes the associated policy terms to XML.

In working with archiving:

- You can modify the code that controls which policy terms the Archive work queue archives. See “Plugin to Determine Archive Eligibility” on page 454.
- You can configure how the Archive writer and worker daemons behave. See “Work Queues” on page 100 in the *System Administration Guide*.
- You can view the current status of the archive process and manually control it. See “Monitoring Archiving Activity” on page 450 for details.

### Archive Statistics

After running the Archive work queue, Guidewire recommends that you update database statistics. The Archive work queue makes large changes to the tables. Updating database statistics enables the optimizer to pick better queries based on more current data.

#### See also

- “Configuring Database Statistics” on page 42 in the *System Administration Guide*.
- “Work Queues” on page 100 in the *System Administration Guide*
- “Configuring Work Queues” on page 105 in the *System Administration Guide*

## Archiving Plugins

If you implement archiving, you must implement the following:

- Plugin to Determine Archive Eligibility
- Plugin to Store and Retrieve Archived Data

### Plugin to Determine Archive Eligibility

If you implement archiving, then you must implement an archiving plugin to determine when to archive a policy term. In the base configuration, Guidewire provides a demonstration plugin, `IPCArchivingPlugin`, as an example. This implementation includes:

- The `IPCArchivingPlugin` plugin – `gw.plugin.archive.impl.PCArchivingPlugin`

The work queue called `ArchivePolicyTermWorkQueue` triggers the actual archiving. This work queue runs the `ArchivePolicyTerm` batch process.

### Plugin to Store and Retrieve Archived Data

If you implement archiving, then you must implement an archive source plugin to store and retrieve from the backing store. In the base configuration, Guidewire provides a demonstration plugin, `IArchiveSource`, as an example of how to implement an archive source plugin. This implementation includes:

- The `IArchiveSource` plugin – `gw.plugin.archive.impl.PCArchivingRecordingPlugin`

After archiving the policy period, this plugin removes any `UWIssueHistory` instances in the active database that were auto-approved.

- Its superclass – `gw.plugin.archiving.ArchiveSource`

It is possible for `IArchiveSource` method calls to occur both inside and outside of an archive transaction, depending on the method. For retrieve, being outside the transaction means that if the retrieve fails, and you made updates during the method call, then PolicyCenter *still* persists that update to the database.

**IMPORTANT** Guidewire provides the `IArchiveSource` plugin implementation for demonstration purposes only. Guidewire expects you to implement an archive source plugin that meets your specific business needs. Any archive source plugin that you create must implement the `IArchiveSource` interface.

#### See also

- “Archiving Integration” on page 555 in the *Integration Guide*

## Archiving Test Tool

PolicyCenter includes the **Archiving Test** tool to help you test archiving on a job or policy term. Use this tool during development to see the effect of the Archive Policy Terms batch process on a selected job or policy term. This tool also has a link to run the Archive Policy Terms batch process.

To access this tool, press ALT+SHIFT+T and select **Internal Tools** → **Archiving Test**.

## Flushing Other Work Queues

A policy term cannot be archived if it is associated with one or more workflows. To more effectively archive policy terms, clean up workflows before archiving.

In **Flush other work queues (Purge workflows, etc.)**, click **Run**.

This command cleans up workflows by running the following batch processes:

- Purge Workflow
- Purge Workflow Logs
- Workflow

## Archive Term by Job Number

Follow these step-by-step instructions to archive a policy term by entering the job number in the **Archiving Test** tool.

1. Enter the **Job id** of the policy term you wish to archive.
2. To archive the policy term without checking to see if the policy term is eligible for archiving, select **Skip validation (archive even with open jobs, etc.)**.

If you did not choose to skip validation, then the policy term is archived based on the current configuration, including the `IPCArchivingPlugin` plugin implementation.

If you choose to skip validation, then the policy term is archived regardless of whether there are open jobs or whether the server time has reached the `PolicyTerm.NextArchiveCheckDate` date.

3. Click **Archive** to archive the policy term with that job id.

PolicyCenter does not run the Archive Policy Term batch process but performs equivalent actions. The tool accesses the configuration parameters and runs the `IPCArchivingPlugin` plugin implementation.

## Archive Policy Term by Policy Number and Term Number

Follow these step-by-step instructions to archive a policy term by the policy number and term number using the **Archiving Test** tool.

1. Enter the **Policy number** and **Term number** of the policy term you wish to archive.
2. To archive the policy term without checking to see if the policy term is eligible for archiving, select **Skip validation (archive even with open jobs, etc.)**.  
If you did not choose to skip validation, then the policy term is archived based on the current configuration, including the `IPCArchivingPlugin` plugin implementation.  
If you choose to skip validation, then the policy term is archived regardless of whether there are open jobs or whether the server time has reached the `PolicyTerm.NextArchiveCheckDate` date.
3. Click **Archive** to archive the policy term associated with the policy and term numbers.

PolicyCenter does not run the Archive Policy Term batch process but performs equivalent actions. The tool accesses the configuration parameters and runs the `IPCArchivingPlugin` plugin implementation.

## Running the Archive Policy Terms Batch Process from the Tool

You can run the Archive Policy Terms batch process from the **Archiving Test** tool.

In **Run archiving batch process**, click **Run**. This is the same as running the Archive Policy Terms batch process from the **Server Tools** → **Batch Process Info** screen.

# Configuring Quote Purging

Quote purging provides batch processes to purge jobs that did not result in bound policies and prune alternate policy periods created through multi-version quoting and side-by-side-quoting. Quote purging also removes orphaned policy periods. Purging and pruning removes these jobs, policy periods, and associated objects from the PolicyCenter database. Quote purging is not a reversible operation. Quote purging is disabled in the base configuration.

This topic includes:

- “Quote Purging Configuration Overview” on page 457
- “Quote Purging Object Model” on page 458
- “Quote Purging Batch Processes” on page 461
- “Using Events to Notify External System of Purged or Pruned Entities” on page 466
- “Purging Test Tool” on page 466

## Quote Purging Configuration Overview

Quote purging is accessed through the **Purge** and **Purge Orphaned Policy Periods** batch processes. These batch processes make use of configuration parameters and a purge plugin.

You can configure quote purging by making modifications to the following:

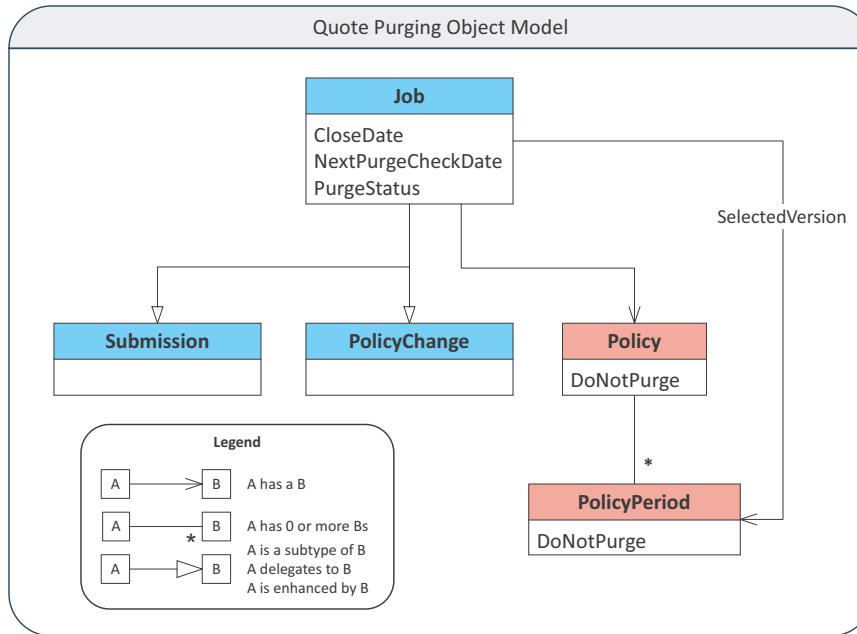
- **Batch processes** – Specify how often, if ever, to run the batch processes that remove jobs and policy periods from the database. For more information, see “Quote Purging Batch Processes” on page 461.
- **Configuration parameters** – You can use these parameters to specify the number of days that must pass before purging a job or pruning a policy period. For more information, see “Quote Purging Configuration Parameters” on page 71.
- **Purge plugin** – Modify quote purging through Gosu code. By configuring the purge plugin, you can control which jobs are purged or pruned. You can also choose to retain certain information related to those jobs. For more information, see “Quote Purging Plugin” on page 174 in the *Integration Guide*.

In addition, you can access quote purging through:

- **Web services** – From an external system, set a flag to prevent a job or policy period from being purged or pruned. For more information, see “Quote Purging Web Services” on page 119 in the *Integration Guide*.

## Quote Purging Object Model

This topic provides information about objects and properties related to quote purging.



In the default configuration, quote purging purges submission and policy change job subtypes.

If the **DoNotPurge** flag on **Policy** is true, then quote purging does not purge the **Job** associated with this **Policy** or prune **PolicyPeriod** objects.

If the **DoNotPurge** flag on **PolicyPeriod** is true, then quote purging does not prune this **PolicyPeriod**.

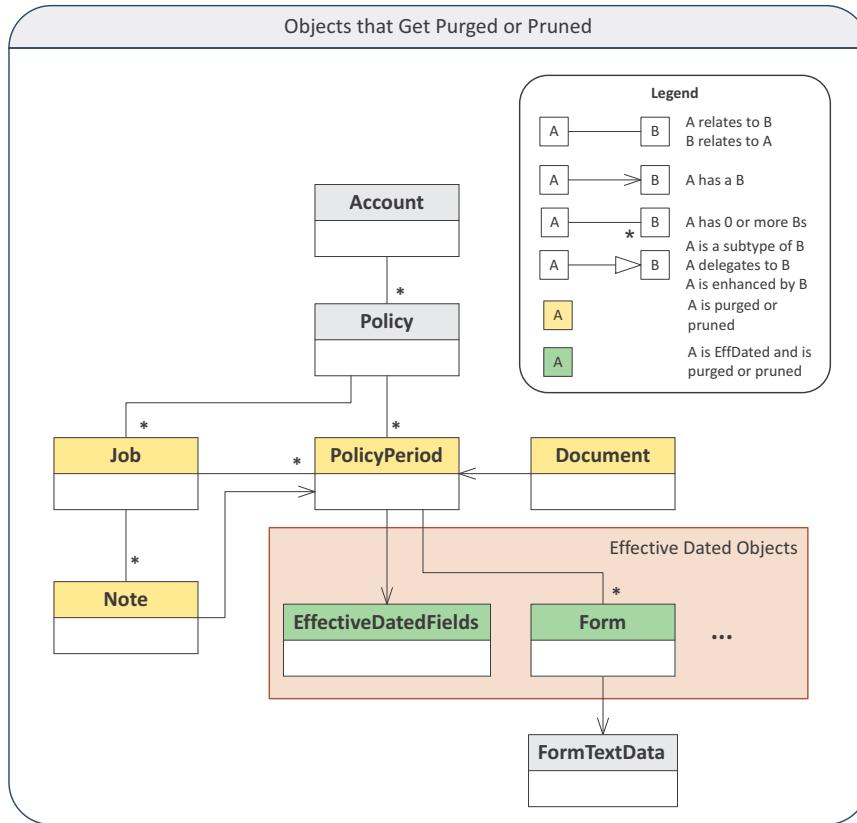
You can manage the **DoNotPurge** property in Gosu code and in the **PurgeAPI** web service. To set the **DoNotPurge** property, you must have the **Enable purging for a policy period** and **Disable purging for a policy period** permissions. The codes for these permissions are **purgeenable** and **purgedisable**, respectively.

### See also

- “Quote Purging Plugin” on page 174 in the *Integration Guide*
- “Quote Purging Web Services” on page 119 in the *Integration Guide*

## Objects that Get Purged or Pruned

The following illustration shows some of the objects that get purged or pruned.

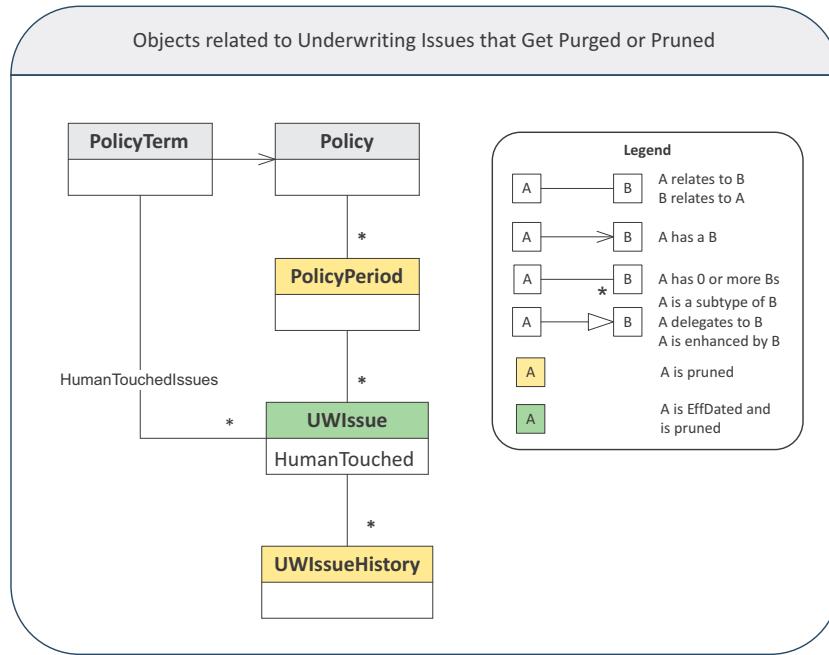


Purging and pruning remove the policy period and all effective dated objects in the policy period branch. Purging and pruning also remove objects connected to the policy period through a foreign key that would otherwise point to a removed object. These objects directly related to the policy period include notes, documents, activities, and forms. Purging and pruning does not remove objects not directly related to the policy period.

Because form text data can be shared between policy periods, quote purging does not remove form text data. As a result, form text data can be orphaned as a result of quote purging. The Form Text Data Delete batch process removes orphaned form text data. For more information, see “Scheduling Work Queues and Batch Processes” on page 107 in the *System Administration Guide*.

Only purging removes the job. If purging removes all jobs associated with a policy, it also removes the **Policy** object, for example, when a **Policy** only exists because of an unbound submission. By definition, pruning only removes policy periods.

The following illustration shows objects related to underwriting issues that get purged or pruned.



Purging and pruning also remove objects related to underwriting issues on the policy period. Human-touched underwriting issues are not purged or pruned. A human-touched `UWIssue` has the `HumanTouched` property set to true.

## Quote Purging Batch Processes

Quote purging purges jobs and prunes policy periods, and purges orphaned policy periods. You access quote purging through batch processes. The quote purging batch processes are:

Name	Code	Description
Purge	Purge	Purges jobs and prunes policy periods that meet the purge and prune criteria. This batch process deletes jobs and other entities from the database. For more information, see "Purge Batch Process" on page 462.
Purge Orphaned Policy Periods	PurgeOrphanedPolicyPeriod	Purges policy periods orphaned as a result of preemption. This batch process deletes policy periods and other entities from the database. For more information, see "Purge Orphaned Policy Periods Batch Process" on page 466.
Reset Purge Status and Check Dates	ResetPurgeStatusAndCheckDates	<p>Reset the purge status and purge or prune dates on the jobs. Run this batch process if and when you have a need to reset the purge status and purge date.</p> <p>For each job, this batch process:</p> <ul style="list-style-type: none"> <li>Sets the Job.PurgeStatus property to Unknown.</li> <li>Sets the Job.NextPurgeCheckDate to null.</li> </ul> <p>For example, if a job has a PurgeStatus of NoActionRequired or Pruned, the batch process resets the PurgeStatus to Unknown.</p>

**IMPORTANT** The Purge and Purge Orphaned Policy Periods batch processes remove jobs and other entities from the database. These batch processes are not reversible operations. The Reset Purge Status and Check Dates batch process is also not reversible.

## Enabling Quote Purging Batch Processes

In the base configuration, the Purge and Purge Orphaned Policy Periods batch processes are disabled. To enable these batch processes, you must do the following in Studio:

1. In config.xml, set the following parameters to true to enable running the batch processes from the Server Tools → Batch Process Info screen in PolicyCenter:

Batch process	Parameter
Purge	PruneAndPurgeJobsEnabled
Purge Orphaned Policy Periods	PurgeOrphanedPolicyPeriodsEnabled

2. To enable the batch process to run on a regular schedule, uncomment the following ProcessSchedule entries in scheduler-config.xml and modify the schedule to meet your needs:

Batch process	Process
Purge	Purge
Purge Orphaned Policy Periods	PurgeOrphanedPolicyPeriod

## Batch Processes to Run Prior to Running the Purge Batch Process

A policy period cannot be purged if it is associated with one or more workflows. To more effectively purge and prune policy periods, clean up workflows before purging or pruning by running the following batch processes:

- Purge Workflow
- Purge Workflow Logs
- Workflow

## Purge Batch Process

The Purge batch process purges jobs and prunes policy periods that meet the purge and prune criteria.

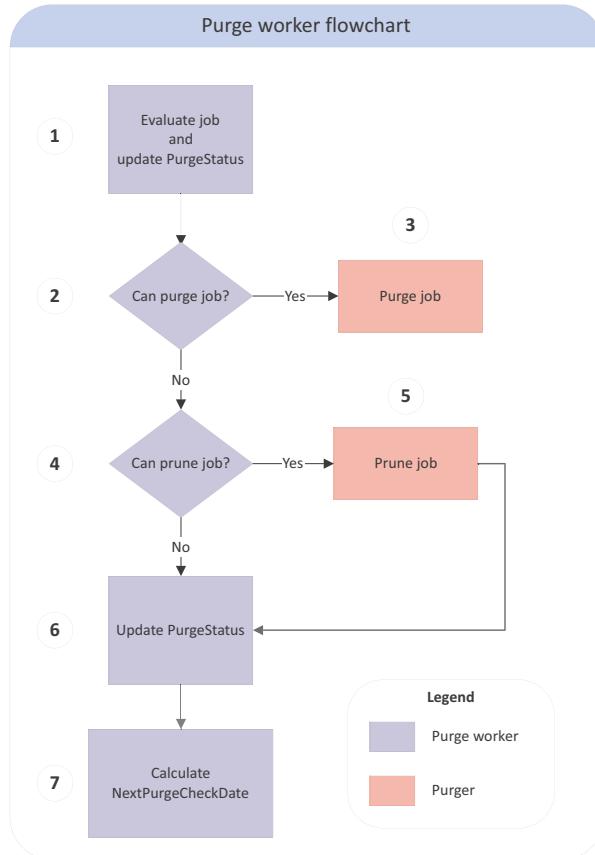
Jobs for which all of the following apply are placed into the Purge work queue:

Condition	Configurable in Purge plugin?	Configurable in config.xml?
Job subtype is allowed for purging or pruning.	●	
The batch process calls the following property getters in the <code>PurgePlugin</code> plugin interface:		
• <code>AllowedJobSubtypesForPurging</code>		
The Job is closed, and the <code>CloseDate</code> is at least <code>PurgeJobsRecentJobCompletionDays</code> or <code>PruneVersionsRecentJobCompletionDays</code> in the past.		
The <code>NextPurgeCheckDate</code> on the job has passed.		
The <code>PurgeStatus</code> has a value other than <code>NoActionRequired</code> .		

To enable this batch process, see “Enabling Quote Purging Batch Processes” on page 461.

## Purge Worker

The Purge worker picks up a job from the Purge work queue and purges or prunes jobs that meet the criteria. The following flow chart shows how the purge worker processes the job. Any job that emerges from step 7 will be considered for purging by the batch process on or after the `NextPurgeCheckDate`.



The numbered items in the flow chart are described below.

- In this step, the job is evaluated according to the following criteria:

Condition	Configurable in Purge plugin?	Configurable in config.xml?
Is the PurgeStatus not equal to <code>NoActionRequired</code> ?		
On the policy period that is the <code>SelectedVersion</code> , is the Status not bound, making the job a candidate for purging?		
Does the job have more than one policy period, making the job a candidate for pruning?		

Then the worker updates the PurgeStatus.

2. This decision point evaluates whether the job and all of its policy periods can be purged. A job can be purged if all of the following apply:

Condition	Configurable in Purge plugin?	Configurable in config.xml?
The policy is not bound.		
The job subtype is a subtype for purging.	●	
Calls the <code>AllowedJobSubtypesForPurging</code> getter in the <code>PurgePlugin</code> plugin interface. See “Get Allowed Job Subtypes for Purging” on page 176 in the <i>Integration Guide</i> .		
The <code>DoNotPurge</code> flag is set to <code>false</code> on the policy and all policy periods on the job.		
The purge date returned by the <code>getPurgeJobDate</code> method is less than or equal to the current date.	●	●
The purge date is calculated by comparing the job close and policy end dates with configuration parameters. For more information, see “Quote Purging Configuration Parameters” on page 71.		
Calls the <code>getPurgeJobDate</code> method in the <code>PurgePlugin</code> plugin interface. See “Get Purge Job Date” on page 177 in the <i>Integration Guide</i> .		
The job has no open activities.		
The job has no archived policy periods.		
There are no workflows on any policy period in the job.		
For information about a process that purges workflows, see “Purging Old Workflows and Workflow Logs” on page 47 in the <i>System Administration Guide</i> .		

3. In this step, the purger purges the job in two separate bundles:

- a. The first bundle calls the `createContext` and `prepareForPurge` methods in the `PurgePlugin` plugin interface. For more information, see “Create Context” on page 175 in the *Integration Guide* and “Prepare for Purge” on page 175 in the *Integration Guide*.

For each policy period in the job, the purger calls the `skipPolicyPeriodForPurge` method in the `PurgePlugin` plugin interface. This bundle then purges the job and the related objects.

- b. The second bundle calls the `postPurge` method in the `PurgePlugin` plugin interface. For more information, see “Take Actions After Purge” on page 176 in the *Integration Guide*.

**IMPORTANT** Guidewire recommends that you put business critical operations in the `prepareForPurge` method so that the first bundle does the purge and business critical operations. Do not put business critical operations in the second bundle. If by chance the system fails while the purger is processing the job, the second bundle may not occur. If the first bundle does not complete, the bundle transaction consisting of the `prepareForPurge` and the purge-from-database action is rolled back. The job will be again be a candidate for purging or pruning the next time the batch process picks it up.

4. This decision point evaluates whether the policy periods on a job can be pruned. The policy periods on a job can be pruned if all of the following apply:

Condition	Configurable in Purge plugin?	Configurable in config.xml?
The job subtype is a subtype for pruning.	●	
Calls the <code>AllowedJobSubtypesForPruning</code> getter in the <code>PurgePlugin</code> plugin interface. See “Get Allowed Job Subtypes for Purging” on page 176 in the <i>Integration Guide</i> .		
The prune date returned by the <code>getPruneJobDate</code> method is less than or equal to the current date.	●	●
The prune date is calculated by comparing the job close and policy end dates with configuration parameters. For more information, see “Quote Purging Configuration Parameters” on page 71.		
Calls the <code>getPruneJobDate</code> method in the <code>PurgePlugin</code> plugin interface. See “Get Prune Job Date” on page 177 in the <i>Integration Guide</i> .		
The job has no open activities.		
The job has no archived policy periods.		

If the previous conditions are met, then each individual policy period is evaluated for pruning. The policy period is sent to the purger if the following apply:

Condition	Configurable in Purge plugin?	Configurable in config.xml?
The policy period is not bound.		
The <code>DoNotPurge</code> flag is set to <code>false</code> on the policy period.		
There are no workflows on the policy period.		

5. In this step, the purger prunes policy periods on the job in two separate bundles:

- The first bundle calls the `prepareForPurge` method in the `PurgePlugin` plugin interface. For more information, see “Prepare for Purge” on page 175 in the *Integration Guide*.  
For each policy period in the job except the selected version, the purger calls the `skipPolicyPeriodForPurge` in the `PurgePlugin` plugin interface. This bundle then purges the policy periods and the related objects.
- The second bundle calls the `postPurge` method in the `PurgePlugin` plugin interface. For more information, see “Take Actions After Purge” on page 176 in the *Integration Guide*.

**IMPORTANT** Guidewire recommends that you put business critical operations in the `prepareForPurge` method so that the first bundle does the purge and business critical operations. Do not put business critical operations in the second bundle. If by chance the system fails while the purger is processing the job, the second bundle may not occur. If the first bundle does not complete, the bundle transaction consisting of the `prepareForPurge` and the purge-from-database action is rolled back. The job will again be a candidate for purging or pruning the next time the batch process picks it up.

6. This step updates the `PurgeStatus` to one of the following values:

- `NoActionRequired` – This is the most common case. For example, `PurgeStatus` gets this value when the only remaining period is bound.
- `Pruned` – When there is one and only one policy period version remaining.

- Unknown – If the purge status is unknown. For example if a version that is not the selected version is marked DoNotPurge, then pruning leaves that version as well as the selected version.
7. This step calculates the next purge check date. This step calls the `calculateNextPurgeCheckDate` method in the `PurgePlugin` plugin interface. For more information, see “Calculate Next Purge Check Date” on page 177 in the *Integration Guide*.
- When the batch process runs again, this job will be reevaluated for purging or pruning.

## Purge Orphaned Policy Periods Batch Process

The **Purge Orphaned Policy Periods** batch process finds *orphaned* policy periods, which are policy periods not associated with a job. Preempted jobs create orphaned policy periods.

For each orphaned policy period, this batch process calls the following method in the `PurgePlugin` plugin interface:

- `skipOrphanedPolicyPeriodForPurge` method that signals not to purge an orphaned policy period. For more information, see “Skip Orphaned Policy Period for Purge” on page 176 in the *Integration Guide*.

**Note:** If you wish to never purge orphaned policy periods, Guidewire recommends that you not enable the **Purge Orphaned Policy Periods** batch process. Customizing this method is not an effective way to never purge orphaned policy periods.

To enable this batch process, see “Enabling Quote Purging Batch Processes” on page 461.

## Using Events to Notify External System of Purged of Pruned Entities

PolicyCenter generates an event when quote purging or pruning removes a `PolicyPeriod`, `Job`, or `Policy` entity instance. You can use messaging to send this event to external systems that track PolicyCenter entities or data. The events that PolicyCenter generates for quote purging are:

- `JobPurged`
- `PolicyPurged`
- `PolicyPeriodPurged`

### See also

- “Messaging and Events” on page 289 in the *Integration Guide*

## Purging Test Tool

The **Purging Test** tool is provided to help you see the effect of purging or pruning on one job. Use this tool during development to see the effect of the Purge batch process on a selected job. This tool also has a link to run the Purge batch process.

To access this tool, press ALT+SHIFT+T and select **Internal Tools** → **Purging Test**.

## Flushing Other Work Queues with the Purging Test Tool

A policy period cannot be purged if it is associated with one or more workflows. To more effectively purge and prune policy periods, clean up workflows before purging or pruning.

- In **Flush other work queues (Purge workflows, etc.)**, click **Run**.

This command cleans up workflows by running the following batch processes:

- `Purge Workflow`

- Purge Workflow Logs
- Workflow

## Purging or Pruning a Job with the Purging Test Tool

Follow these step-by-step instructions to purge or prune a particular job with the Purging Test tool.

1. Enter the work order number of the job you wish to purge.
2. Click **Find/Refresh Job** to fetch the job from the database.

PolicyCenter displays a **Purge or Prune Job** section. This section displays the following information about the job:

Field	Description
Job Number	The number of the job to purge or prune.
Subtype	The job subtype.
Close Date	The job close date.
DoNotPurge flag (Policy)	The value of the DoNotPurge flag on Policy.
Policy Period(s)	The number of policy periods associated with this job. The <b>Prune -- Purge Policy Period(s)</b> section displays more information about each policy period.
Coverage End	The coverage end date of the policy.

You can select to purge or prune the selected job.

3. To purge or prune the job without checking to see if the job is eligible for purging or pruning, select **Skip configurable Purging checks** or **Skip configurable Pruning checks**.

If you did not choose to skip checks, then the job is purged or pruned based on the current configuration, including the **PurgePlugin** implementation.

If you choose to skip checks, then the job is purged or pruned regardless of the **Subtype**, **Close Date**, or **Coverage End**.

If you choose to skip checks, a job can be purged or pruned only if the following are true:

- The job's **CloseDate** is not null.
- The **DoNotPurge** flag is not set on the policy associated with the job.
- The job has no open activities.
- The job is not archived.

Additionally, if you choose to skip checks, the following criteria also apply. For purging, each policy period on the job must meet this criteria. For pruning, the policy period being pruned must meet this criteria.

- The **PolicyPeriod** is not bound.
- The **DoNotPurge** flag on the **PolicyPeriod** is not set.
- The **PolicyPeriod** has no workflows.

4. Click **Purge Job** or **Prune Job** to purge or prune the selected job.

PolicyCenter does not run the Purge batch process but performs equivalent actions to purge or prune the selected job. The purge or prune accesses the configuration parameters and runs the **PurgePlugin** implementation.

## Pruning Policy Periods with the Purging Test Tool

After selecting a job with the **Purging Test** tool, the policy periods attached to the selected job appear in the **Prune -- Purge Policy Period(s)** section. This section displays the following information about each policy period:

Field	Description
Policy Period	The version number of the policy period, and whether it is the selected policy period.
Period Start	The policy period start date.
Period End	The policy period end date.
Period Status	The policy period status.
DoNotPurge (period)	The value of the DoNotPurge flag.
Flip DoNotPurge bit	A button to toggle the DoNotPurge flag on this policy period.

1. To prune policy periods without checking to see if the policy period is eligible for pruning, select **Skip checks for purging policy period**.

If you choose to skip checks, then the policy period is pruned regardless of the **Subtype**, **Close Date**, or **Coverage End**.

If you did not choose to skip checks, then the policy period is pruned based on the current configuration, including the **PurgePlugin** implementation.

2. Click **Purge** in the row of the policy period you wish to prune.

The **Purge** button is enabled for unselected policy period versions.

PolicyCenter does not run the **Purge** batch process, but performs equivalent actions to prune the selected policy period. The **purge** or **prune** accesses the configuration parameters, and runs the **PurgePlugin** implementation.

## Running the Purge Batch Process from the Purging Test Tool

You can run the **Purge** batch process from the **Purging Test** tool.

In **Run Purge Batch Process**, click **Run**. This is the same as running the batch process from the **Server Tools → Batch Process Info** screen. The batch process purges jobs and prunes policy periods in the PolicyCenter database.

# Configuring Underwriting Authority

*Underwriting authority* in PolicyCenter provides an underwriting infrastructure which you can modify or augment. This topic describes how to configure underwriting authority.

This topic includes:

- “Overview of Configuring Underwriting Authority” on page 469
- “Implementing Underwriting Authority” on page 471
- “Configuring Authority Grants” on page 478
- “Configuring Underwriting Issues” on page 479
- “Underwriting Issue Type System Table” on page 489
- “Configuring Approvals” on page 495
- “Handling Underwriting Issues in Policy Revisions” on page 497

**See also**

- “Underwriting Authority” on page 415 in the *Application Guide*
- “Authority Profiles” on page 669 in the *Application Guide*
- “Configuring Underwriting Authority and Multicurrency” on page 582

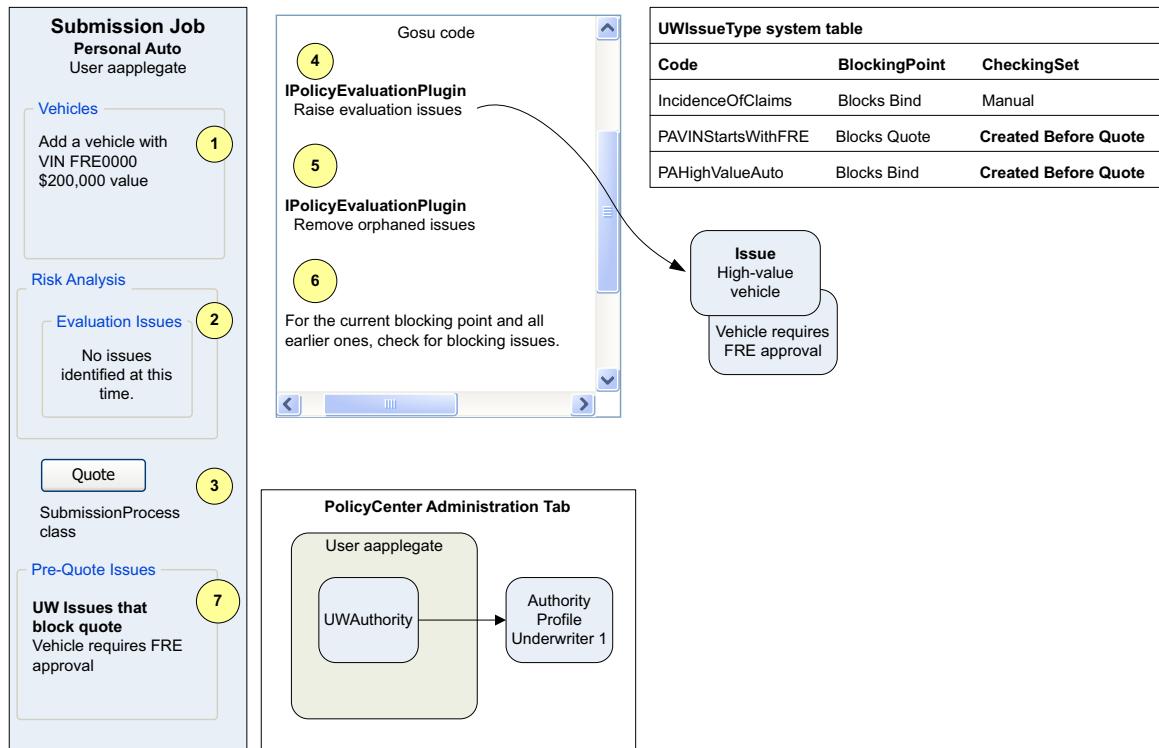
## Overview of Configuring Underwriting Authority

This topic provides a high-level description of how issues work in the default configuration.

- “Raising Underwriting Issues” on page 470
- “Issues Blocking Progress” on page 471

## Raising Underwriting Issues

This illustration shows a simplified version of how PolicyCenter raises underwriting issues. In a submission job, the user adds a high value vehicle with a VIN number. The high value vehicle causes PolicyCenter to raise underwriting issues before quoting.



1. The user aapplegate, who has an Underwriter 1 authority profile, starts a personal auto submission job. On the Vehicles screen, she adds a vehicle with VIN FRE0000 valued at \$200,000.
2. The user advances to the Risk Analysis screen. PolicyCenter has not identified any issues at this time.
3. The user clicks the Quote button.
4. The IPolicyEvaluationPlugin checks whether there are issues to create before quote for this policy period and its lines of business. For example, the PA: High value vehicle issue type raises an issue if vehicle cost is greater than \$100,000. In this case, the code creates an issue based on the VIN number and another issue for high-value vehicle.

The IPolicyEvaluationPlugin plugin has input parameters for the current checking set (Quote Issues) and policy period. The plugin calls evaluator classes that evaluate and raise underwriting issues. The plugin calls line-of-business-specific evaluators (PA\_UnderwriterEvaluator.gs for example) for each line of business in the policy period. The plugin then calls the default evaluator (DefaultUnderwriterEvaluator.gs). You can modify the evaluator classes to raise underwriting issues for your underwriting issue types.

The evaluators retrieve issue type definitions from the UWIssueType system table (uw\_issue\_type.xml). When determining whether to raise an issue, the evaluators use the system table and the underwriting authority of the current user.

5. The IPolicyEvaluationPlugin checks for orphaned issues and removes them. In this case, the plugin finds no orphaned issues. Orphaned issues are underwriting issues that were generated by this checking set but are no longer an issue at this time.

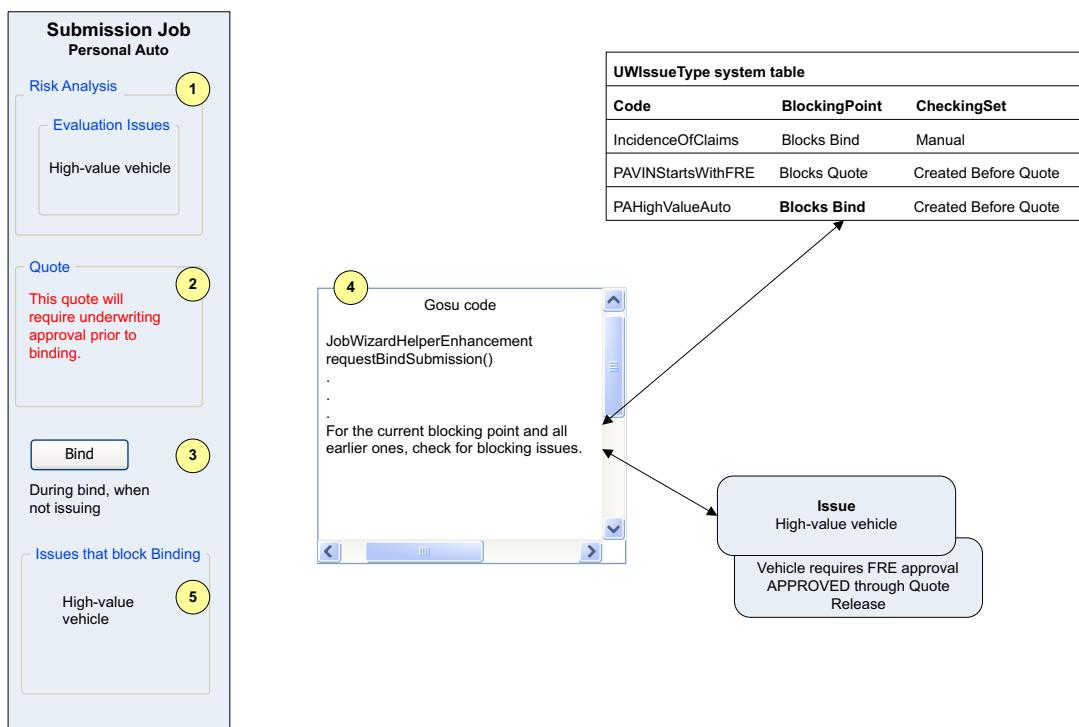
6. Before quoting, the Gosu code checks for issues that block quote and any earlier blocking points. In this example, there are issues that block quote.
7. PolicyCenter displays a screen listing the VIN issue that blocks quote and does not quote the policy. PolicyCenter does not highlight the issue for the high value vehicle because job progress is already blocked at an earlier point.

#### See also

- “Authority Profiles” on page 669 in the *Application Guide*

## Issues Blocking Progress

The following illustration shows what happens when there are issues blocking progress. This topic continues the previous submission job after approval of the VIN issue. There is an issue blocking progress during bind, when not issuing.



1. The **Risk Analysis** screen displays an issue blocking bind for a high-value vehicle.
2. PolicyCenter quotes the policy.
3. The user selects to bind but not issue the policy.
4. The Gosu code checks to see if it needs to raise issues and finds none. It checks for blocking issues at the current blocking point. It references the **UWIssueType** system table and existing issues.
5. PolicyCenter displays a screen with the high-value vehicle issue blocks bind.

## Implementing Underwriting Authority

Implementing underwriting authority requires changing various areas of the PolicyCenter configuration. This topic provides steps to help guide you through the process, and includes instructions for:

- Creating a clear definition of the underwriting issues

- Modifying the code that raises underwriting issues
- Configuring authority profiles and users
- Writing code to create issues
- Adding a checking set
- Adding a value comparator

#### To implement underwriting authority

1. Define the underwriting issues for your implementation. Review the existing underwriting issues to determine if they need to be modified or removed. Determine whether you need to add new underwriting issues. For more information, see “Defining an Underwriting Issue” on page 473.
2. In Product Designer, define underwriting issue types in the `UWIssueType` system table. Remove any issue types that you have not used and will not use in the future.

**IMPORTANT** Guidewire recommends that you do not change the value comparator and auto-approvable flags for an issue type after deployment. When issues of a given type have been created, that issue type must be maintained going forward. For proper management of policies, the authority grants that have these older issues attached to them must also be maintained. However, you can create new issue types to use for future occurrences.

3. In Studio, write or modify code that creates the issues for each type. You can create the issues by line of business or for all lines of business. For more information, see “Creating Underwriting Issues” on page 474.
4. In PolicyCenter, create authority profiles in the **Administration → Authority Profiles** screen. Each authority profile grants authority for specific issues. For more information, see “Authority Profiles” on page 669 in the *Application Guide*.
5. In PolicyCenter, add the authority profiles to individual users in the **UW Authority** tab in the **User** screen. For more information, see “Working with Authority Profiles” on page 671 in the *Application Guide*.
6. In PolicyCenter, grant users permissions as appropriate for their role. The permissions that apply to underwriting issues are:
  - **View risk analysis evaluation issues** – The code for this value is `viewriskevalissues`.
  - **View risk analysis referral reasons** – The code for this value is `viewriskrefreasons`.
  - **Edit Lock Override** – The code for this value is `editlockoverride`.
  - **Quote Hide Override** – The code for this value is `quotehideoverride`.
  - **Reject UW Issues** – The code for this value is `uwreject`.
  - **Reopen UW Issues** – The code for this value is `uwreopen`.
  - **Approve all UW Issues** – This permission grants the ability to approve any issue to any level. It is intended only for resolving missing authority issues in a time-critical situation. The code for this value is `uwapproveall`.

For more information about permissions, see “Security: Roles, Permissions, and the Community Model” on page 647 in the *Application Guide*.

7. Specify the PolicyCenter user that will process automated renewals.

**Note:** Automated processes, such as automated renewals, treat all underwriting issues as auto-approvable. For more information, see “Approvals for Auto-approvable Issues” on page 495 and “Auto-approvable” on page 492.

- a. In your implementation of the renewal plugin, implement the `getAutomatedRenewalUser` method to return this user. In the default configuration, this method returns the user `renewal_daemon`. (Be sure to assign appropriate authority profiles to the `renewal_daemon` user. Guidewire suggests that the authority profiles be similar to other users.)

For details about the renewal plugin, see “Renewal Plugin” on page 165 in the *Integration Guide*.

- b. Add that user in PolicyCenter if necessary. Assign authority profiles to that user. For more information, see “Assigning an Authority Profile to a User” on page 671 in the *Application Guide*.
8. If you will be automatically processing policy changes, specify the user for PolicyCenter to use. You can automatically start policy changes by using the `startAutomaticPolicyChange` method in the `PolicyChangeAPI`. In the default configuration, this method uses the `policychange_daemon` user. For details about this API, see “Policy Change Web Services” on page 114 in the *Integration Guide*.
9. You can also make the following modifications.

  - Add a new checking set and change the job processes to check for it. See “Adding a New Checking Set” on page 475.
  - Add a new value comparator to use with issues. See “Adding a New Value Comparator” on page 476.
  - Add a new value formatter to use with issues. See “Adding a New Value Formatter” on page 477.

## Defining an Underwriting Issue

This topic helps you define the underwriting issues for your implementation. You need to determine whether to modify or remove existing underwriting issues. You need to determine whether to add new underwriting issues.

### To define an underwriting issue

**1. What is the clear definition of this underwriting issue type?** Is this an issue because the driver is more than 80 years old? Is this an underwriting issue because the business prepares fried foods?

**2. How will you detect when the policy period has this underwriting issue?** For example, which fields on the `PolicyPeriod` determine whether this underwriting issue is present on the policy? What programming logic will be needed?

**3. What type of value is associated with this underwriting issue?** Does the value affect which underwriters can approve that issue?

A value can be a number, a currency amount, a jurisdiction or set of jurisdictions, a class code or a range of class codes. It can also be another type of value that can be easily represented as a string.

An underwriting issue does not need a value if that value would not affect which underwriters will be able to approve it. For example, suppose you have an underwriting issue type is *Driver is under 25 years old*. If the same set of underwriters can approve the issue whether the age of the driver is 17 or 24, then a value is not needed. However, suppose some underwriters can approve the issue if the age of the driver is at least 21, but others can approve the issue when age is only 16. In this case, use the age as the value associated with the underwriting issue.

You can configure a value for an underwriting issue type even if it is not required. In the first case above, while the age is not strictly necessary, there is no harm in associating it with an underwriting issue. In the default configuration, the underwriter will see the value when PolicyCenter displays the underwriting issue.

For more information, see “Value Formatter” on page 490.

**4. If there is a value associated with this underwriting issue, can you use one of the existing value comparator types?**

In the default configuration, PolicyCenter provides value comparators for numeric values and for jurisdictions. You can add additional comparators if necessary. Adding a new value comparator is complex. Therefore, consider carefully before adding a new one.

For more information, see “Comparing Issue Values” on page 488 and “Comparator” on page 489.

**5. If there is a value associated with this underwriting issue, can you use one of the existing value formatter types?**

In the default configuration, PolicyCenter provides value formatters for integer, currency, jurisdictions and sets of jurisdictions, and raw string values. If these formatters are not sufficient, you may need to add a new value formatter. Although not as complex as adding a new comparator, consider carefully before adding a new one.

For more information, see “Value Formatter” on page 490.

#### **6. On which job types and at which job lifecycle transitions will this underwriting issue be checked for and raised?**

Determine the job types (such as submission, renewal, issuance) that will check for and raise this underwriting issue. Within each the job, determine which lifecycle transitions (such as quoting or binding) check for the underwriting issue. Does PolicyCenter check for this underwriting issue at multiple points or at one point only? The checking set for the underwriting issue determines when PolicyCenter checks for the issue.

Underwriting issues that will not change after a job is quoted may be checked for at only one point, and may be checked for as early as quoting. There is no reason to ever try to check for the underwriting issue again.

Other underwriting issues may require multiple checking sets. For example, suppose PolicyCenter receives payment delinquency information from a billing system. You can raise the corresponding underwriting issue at the next lifecycle point (quoting, binding, issuance) no matter when it is checked for.

**Note:** In general, Guidewire recommends that you raise underwriting issues as early as possible.

For more information, see “Checking Sets and Evaluators” on page 481 and “Checking Set” on page 491.

#### **7. What progress does this underwriting issue block?**

Carefully consider the point at which the underwriting issue blocks progress. For example, you can raise fire hazard issues before quoting a policy, but have the issue block binding. By raising underwriting issues early, the user is aware of the issues early and can plan how to deal with them.

For more information, see “Blocking Points” on page 482 and “Blocking Point” on page 491.

#### **See also**

- “Underwriting Issue Type System Table” on page 489

## [Creating Underwriting Issues](#)

This topic describes how to write the code that creates underwriting issues.

The `IPolicyEvaluationPlugin` is the starting point for evaluating whether to create an underwriting issue. In the base configuration, the `IPolicyEvaluationPlugin` interface is implemented by the `PolicyEvalPlugin` plugin. This plugin creates a `PolicyEvalContext` object and then evaluates (`evaluator.evaluate` method) whether to raise issues for the current checking set and policy lines associated with the policy period.

The plugin uses a default class (`DefaultUnderwriterEvaluator.gs`) and line-of-business-specific classes (`PA_UnderwriterEvaluator.gs` for example) to determine whether or not to raise underwriting issues.

When you evaluate each checking set, your implementation must raise issues and refresh existing issues with current values. Your implementation must also remove orphaned issues. Guidewire recommends that your implementation use the `PolicyEvalContext` class to create and remove orphaned issues because these are complex and error-prone tasks.

This topic provides instructions for creating underwriting issues.

#### **See also**

- “Policy Evaluation Plugin” on page 164 in the *Integration Guide*

### To create underwriting issues

1. To raise underwriting issues for all lines of business, modify the default evaluator class, `gw.lob.common.DefaultUnderwriterEvaluator.gs`. To raise underwriting issues for a particular line of business, modify the evaluator class for that line of business, `gw.lob.pa.PA_UnderwriterEvaluator.gs`, for example.
2. Find the appropriate method for the checking set. For example, to raise underwriting issues for the Renewal checking set, add code in the `onRenewal` method of the evaluator class. You can view and modify the correspondence between checking set and method in the `AbstractUnderwriterEvaluator` class.
3. Use the `PolicyEvalContext.addIssue` method to add underwriting issues. To raise underwriting issues, you need to write Gosu code which examines the `PolicyPeriod` and determines whether to raise one or more issues of this type. For example, the code may raise multiple issues if the `PolicyPeriod` entity contains multiple items of the same type, such as cars or buildings.
4. Use the `PolicyEvalContext.removeOrphanedIssues` method to remove orphaned underwriting issues. The `removeOrphanedIssues` method is called in the `PolicyEvalContext` plugin.

### To raise an underwriting issue

1. Add an `addIssue` method (using one the available method signatures). For example:

```
addIssue(issueType, key, shortDescriptionBlock, longDescriptionBlock, value)
```

2. Provide values for `issueType` and `key`:

<code>issueType</code>	Code of the issue type, use a value from <code>UWIssueType</code> system table
<code>key</code>	IssueKey for the <code>UWIssue</code>

3. Provide a block for the `shortDescription` and `longDescription` parameters (one each). This allows the method to localize the description text for all available locales. (PolicyCenter performs the localization as it evaluates the display key.) Use this format:

```
shortDescriptionBlock = \ -> shortDescriptionDisplayKey
longDescriptionBlock = \ -> longDescriptionDisplayKey
```

4. Enter the `value`, which is the value that PolicyCenter checks to determine whether to raise the issue.

### Example

The following code is an example of how to create an `addIssue` method:

```
policyEvalContext.addIssue("PANumberOfVehicles", "PANumberOfVehicles",
 \ -> displaykey.UWIssue.PersonalAuto.TooManyVehicles.ShortDesc,
 \ -> displaykey.UWIssue.PersonalAuto.TooManyVehicles.LongDesc(numberOfVehicles),
 numberOfVehicles)
```

## Adding a New Checking Set

The default configuration of PolicyCenter provides a number of predefined checking sets. Each checking set is evaluated at specific blocking points for each job type, and PolicyCenter determines whether or not to raise an underwriting issue. For each checking set, you can modify when the checking set is evaluated. However, you will need to add a new checking set if you have an underwriting issue type that requires checking at a new combination of job type and blocking point.

The blocking point at which each checking set is evaluated is specified in the `JobProcessUWIssueEvaluator.gs` class. The code specifies which checking sets are evaluated at all blocking points. For each blocking point, the code specifies which additional checking sets are evaluated. Each job type can also specify when checking sets are evaluated.

For more information about checking sets and blocking points, see:

- “Checking Sets and Evaluators” on page 481

- “Blocking Points” on page 482
- “Job Interactions with Underwriting Issues” on page 483

#### To add a new checking set affecting all job types

If a checking set will be evaluated at the same blocking points for all job types, then you can change the defaults in `JobProcessUWIssueEvaluator`.

1. In Studio, open the `UWIssueCheckingSet` typelist.
2. Add a new typecode entering a code, name, description, and optional priority.
3. In Studio, open `gw.job.JobProcessUWIssueEvaluator.gs`.
4. Modify the static function that returns a `JobProcessUWIssueEvaluator` associated with the specific job type or types that you wish to affect. An example in the default configuration is the `forRenewal()` method. This method overrides the built-in checking at the Quote and Issuance blocking points. (Both the `CheckedAtAllBlockingPoints` as well as the blocking-point-specific lists are examined.)
5. If `JobProcessUWIssueEvaluator` has been modified in a way that affects the `CheckedAt` properties or logic, then you may need to make modifications. Do a similar review of `JobProcess` and how it may call `JobProcessUWIssueEvaluator` or otherwise affect the `CheckedAt` behavior.

#### To add a new checking sets affecting a single job type

1. In Studio, open the `UWIssueCheckingSet` typelist.
2. Add a new typecode entering a code, name, and description.
3. Open `gw.job.JobProcessUWIssueEvaluator.gs`.
4. In the `forJobtype` method, instantiate a new `JobProcessUWIssueEvaluator` with the appropriate `CheckedAt` properties set. Assign it to `this.JobProcessEvaluator`. If this code already exists, you can add a `CheckedAt` property.

For example, the `forSubmission` method in the default configuration is as follows:

```
static function forSubmission() : JobProcessUWIssueEvaluator {
 return new JobProcessUWIssueEvaluator() {
 :CheckedAtAllBlockingPoints = {"All"}
 }
}
```

## Adding a New Value Comparator

You may need to add a new value comparator if:

- **The issue type needs to have an associated value** because different PolicyCenter users have authority to approve specific values or ranges of values. For example, issues of a certain type have an associated class code. PolicyCenter users can be authorized to approve issues within certain class code ranges. There is no existing value comparator for this type. Therefore, you need to create a new value comparator for this type, and implement Gosu code that determines whether one class code or range contains another class code range.
- **The existing comparators do not meet your needs** because:
  - **There is no comparator for this kind of data** – For example, you need a comparator that determines the manufacturer of a car based on the Vehicle Identification Number (VIN).
  - **There is no comparator that compares two values in the way that you require** – For example, there is a state set comparator. However, you need a comparator which determines if a jurisdiction is in the northern, southern, eastern, or western part of the country.

Comparators are used to determine whether one value associated with some `UWIssueType` is within the bounds set by some other value. The comparator is used for two different comparisons:

- Ensuring that an approval value is within a grant value.

- Ensuring that an issue value is within an approval value.

Each new value comparator has a typekey in the `ValueComparator` typelist and a wrapper. There must be exactly one `UWIssueValueComparatorWrapper` for each `ValueComparator` and one `ValueComparator` for each `UWIssueValueComparatorWrapper`. When you instantiate the wrapper, you specify the associated `ValueComparator`. There is a two-way association between typekey and wrapper.

For more information, see “Comparing Issue Values” on page 488 and “Comparator” on page 489.

#### To add a value comparator

- In Studio, edit the `ValueComparator` typelist.
- Add a new typekey entering a code, name, and description.
- Instantiate a wrapper for your new typekey.

In the default configuration, the wrappers are instantiated in the `UWIssueValueComparatorWrapper` class located in the `gw.job.uw` package. Add code to instantiate a wrapper for your new typekey. Examples in the default configuration can be seen at the top of the file:

```
// The wrappers for the various OOTB ValueComparators
static final var _ge : UWIssueValueComparatorWrapper as readonly NumericGEWrapper
 = new NumericGEValueComparatorWrapper("Numeric_GE")
```

If you prefer not to instantiate the wrapper in `UWIssueValueComparatorWrapper`, you can instantiate the wrapper in the location of your choice. However, the wrapper must be instantiated in a class that loads before PolicyCenter calls `UWIssueValueComparatorWrapper.wrap`. For example, you can instantiate your wrappers in a startable plugin which runs after loading the database, but before serving any web pages. For more information, see “Startable Plugins Overview” on page 259 in the *Integration Guide*.

**IMPORTANT** Failure to guarantee that PolicyCenter instantiates the wrapper before loading `UWIssueValueComparatorWrapper` may result in non-deterministic null pointer exceptions if `UWIssueValueComparatorWrapper` returns null for an extended typekey.

- Instantiate a new extension to `UWIssueValueComparatorWrapper` for your typekey in the `gw.job.uw.comparators` package. You can use existing comparators as an example.

## Adding a New Value Formatter

The default configuration of PolicyCenter provides formatters for currency, numbers, jurisdictions, age, units, and U.S. dollars. There is also a formatter that does not format the data. If the formatter you need is not provided in the default configuration, you can add new value formatters. You can also modify the existing value formatters. For more information, see “Value Formatter” on page 490.

#### To add a new value formatter

- In Studio, edit the `ValueFormatterType` typelist.
- Add a new typekey entering a code, name, and description.
- Instantiate the formatter and link it to your new typekey. Write code to format the value.

In the default configuration, the value formatters are instantiated in the `ValueFormatter` class located in the `gw.job.uw` package. Add code to instantiate the value formatter for your new typekey. Also add code to format the value. For example, the following is the code for the `Integer` value formatter in the `ValueFormatter` class:

```
final public static var FOR_INTEGER : ValueFormatter = new ValueFormatter("Integer") {
 override function format(value : String) : String {
 return formatInteger(value)
 }
}
```

The code `ValueFormatter = new ValueFormatter("Integer")` links the formatter to the `Integer` typekey. The `formatInteger` method formats the value.

If you prefer not to instantiate the value formatter in the `ValueFormatter` class, you can instantiate it in the location of your choice. However, the value formatter must be instantiated in a class that is guaranteed to load before PolicyCenter calls the `ValueFormatter.format` method. For example, you can instantiate your wrappers in a startable plugin which runs after loading the database, but before serving any web pages. For more information, see “Startable Plugins Overview” on page 259 in the *Integration Guide*.

---

**IMPORTANT** Failure to guarantee that the value formatter is instantiated before loading `ValueFormatter` may result in non-deterministic null pointer exceptions if `ValueFormatter.format` returns null for an extended typekey.

---

## Configuring Authority Grants

Authority profiles give users the authority to approve particular types of issues. Authority profiles contain authority grants. Users can have more than one authority profile, and authority profiles can be shared among multiple users.

Like roles and permissions, authority grants are always additive. A user can approve an issue if the user has at least one grant that allows them to approve that issue. For example, a user has one profile that grants a user authority to approve total premium up to \$1000. This user also has another profile that grants them the ability to approve total premium up to \$1500. In this case, the \$1500 limit will be the one that is in effect.

An authority grant has an associated value if the comparator specifies a value type other than `None`. If the authority grant for a given issue has an associated value, the grant can either be contingent on the given value or can apply to any possible value. If the grant applies to any value then the `ApproveAnyValue` bit is set on the associated authority grant. Otherwise, PolicyCenter sets the `Value` field on the authority grant, and compares the grant value to the issue value to determine if the issue can be approved. In addition, PolicyCenter compares the grant value to the approval value, to make sure that the approval falls within the limits of the grant.

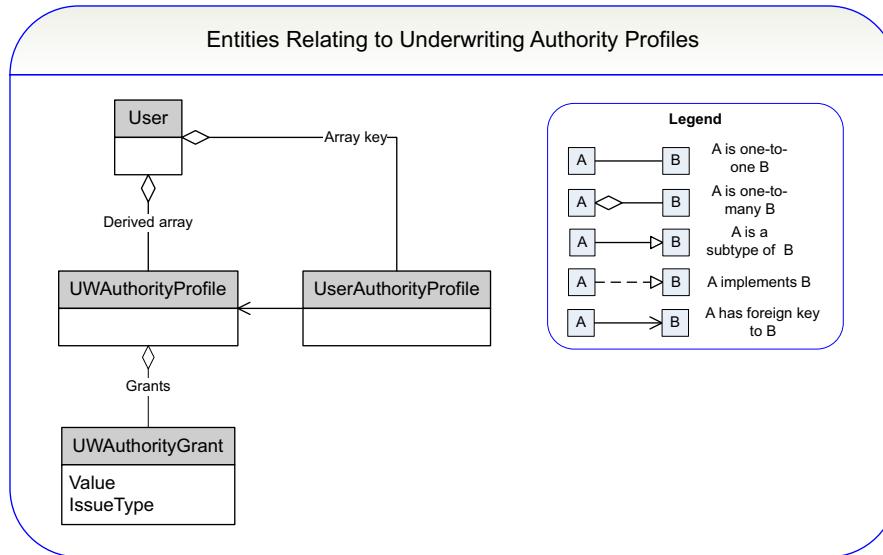
### See also

- “Underwriting Issue Type System Table” on page 489 for more information about issue type definitions used by authority grants.
- “Authority Profiles” on page 669 in the *Application Guide* for more information about creating authority profiles in the PolicyCenter application and how to work with them.

## Underwriting Authority Profile Object Model

Authority profiles contain authority grants. PolicyCenter stores authority profiles as `UWAuthorityProfile` entities, which are made up of `UWAuthorityGrants`. You can access authority profiles from the `User` entity through the `UserAuthorityProfiles` field. The relationship is many-to-many, such that users can have more than one profile, and multiple users can share the same profile.

The following illustration shows the relationship between key entities for underwriting authority profiles.



The following table describes entities associated with authority profiles.

Entity	Description
User	The PolicyCenter user.
UWAuthorityProfile	This entity contains a series of UWAuthorityGrant entities. This entity is similar to a Role entity which serves as a collection of RolePrivilege entities. Although this entity has a Grants array to UWAuthorityGrant, each profile can have at most one grant for a given issue type. Profile names must be unique.
UWAuthorityGrant	This entity represents a grant of authority to approve a particular type of issue. For issues that have an associated value type, that authority can either be restricted to a particular value or can be unrestricted.
UserAuthorityProfile	The authority profile contains a foreign key to UWAuthorityProfile.

## Displaying Authority Grants in the User Interface

The user configures authority grants on the **Authority Profiles** screen under the **Administration** tab. The second column displays value of the comparator column in the **UWIssueType** system table.

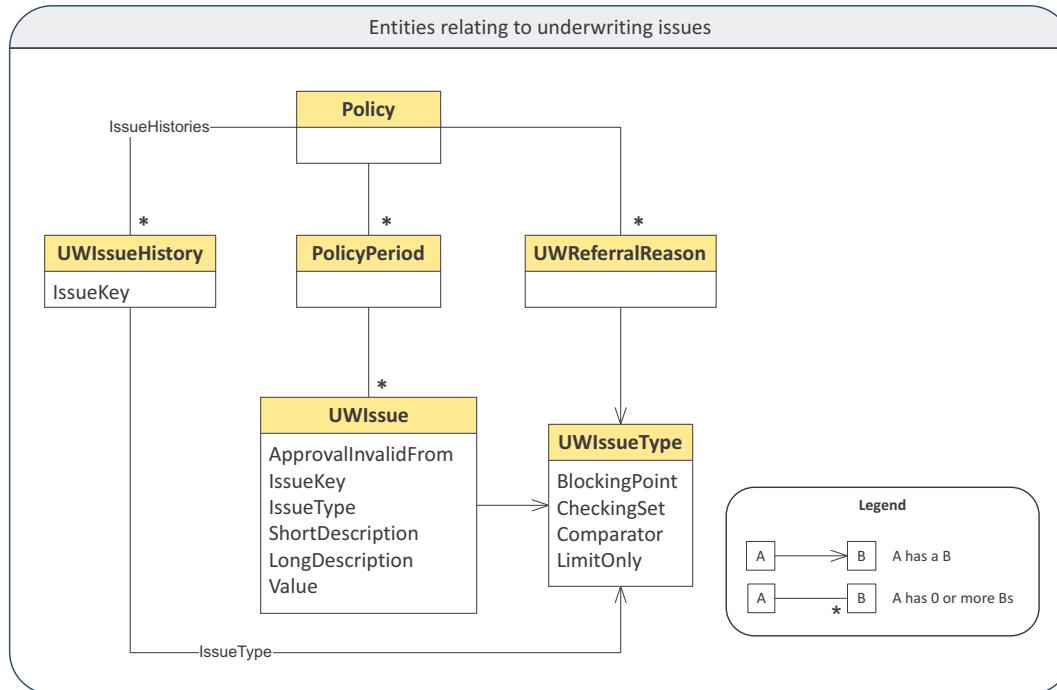
**Note:** Guidewire chose to display the comparators as the strings **At most** and **At least** rather than as the **≤** and **≥** symbols (U+2264 and U+2265, respectively). Guidewire made this choice because databases that do not support Unicode characters may not store these characters correctly. These symbols may be used in Unicode-compliant installations.

## Configuring Underwriting Issues

Issues are at the core of the underwriting authority feature. PolicyCenter raises issues at specific points in the job flow, and issues block progress until they are approved. The two core components of issues are issue types and the code that create issues of those types.

## Underwriting Issue Object Model

The following illustration shows the relationship between key entities for underwriting issues.



The following table describes some of the entities associated with underwriting issues.

Entity or object	Description
PolicyPeriod	<p>Stores information for a specific period of a policy.</p> <p>The <code>UWIssuesIncludingSoftDeleted</code> array provides access to all issues on this policy, including those marked inactive.</p>
UWIssue	<p>These are the issues that evaluator classes create. The issues link to issue types and can contain an approval. These entities can vary in effective time and are copied from branch-to-branch, just like any other effective-dated entity. Some fields on this entity are:</p> <ul style="list-style-type: none"> <li>IssueKey and IssueType – These two fields uniquely identify the issue.</li> <li>ShortDescription – Description of the issue that appears in the user interface.</li> <li>LongDescription – Long description of the issue that appears in the user interface. Some users may have permission to view the short description but not the long description.</li> <li>Value – The value, if any, associated with this issue. If present, the value is compared with authority grants to determine if the user can approve this issue. The value is also compared with approvals to determine if the approval still applies.</li> </ul>
UWIssueHistory	<p>This entity represents a history of issues and approvals on the policy. The <code>IssueKey</code> field is a reference to the <code>UWIssue</code> with that <code>IssueKey</code>.</p> <p>PolicyCenter maintains the history in a set of non-effdated entities on the policy and outside of the context of any particular job or policy period. PolicyCenter maintains this history for the following reasons:</p> <ul style="list-style-type: none"> <li>The entity captures events that happen on branches that do not end up binding.</li> <li>The entity persists after issues are removed or approvals expire.</li> <li>The entity persists even if the related periods are archived.</li> </ul> <p>Since this entity attaches directly to the policy, it is retireable rather than effdated.</p>

Entity or object	Description
UWIssueType	This entity defines issue types. Use Studio to view and edit these entities in the UWIssueType system table. See “Underwriting Issue Type System Table” on page 489.
UWReferralReason	<p>This entity is similar to the UWIssue entity but exists off of the policy rather than the policy period. This entity is used for marking a policy issue outside of a job. It identifies the details of a corresponding UWIssue that will be generated the next time the application checks for UWIssues. Because of the close relationship between a UWReferralReason and the associated UWIssue, the default user interface displays them in a similar way.</p> <p>For more information, see Working With “Working with Underwriting Referral Reasons” on page 420 in the <i>Application Guide</i> and “Configuring Underwriting Referral Reasons” on page 483.</p>

The actual link between User and UWAuthorityProfile is through the array key from User to UserAuthorityProfile then the foreign key from that entity to UWAuthorityProfile. For convenience, there is a derived array from User to UWAuthorityProfile.

## Checking Sets and Evaluators

Checking sets represents points in the job at which issues can be raised. For each checking set, evaluator classes determine whether or not to create issues. Each issue type has a checking set property. You define issue types in the UWIssueType system table. For more information, see “Checking Set” on page 491, “Blocking Points” on page 482, and “Job Interactions with Underwriting Issues” on page 483.

The IPolicyEvaluationPlugin triggers policy evaluation. This plugin is where PolicyCenter adds underwriter issues on a policy period. In the base configuration, this plugin calls evaluator classes to raise issues. The plugin then removes orphaned issues. Orphaned issues are underwriting issues that were generated at a checking set but are no longer an issue when that checking set runs at a later time.

If you need to change this plugin, see “Policy Evaluation Plugin” on page 164 in the *Integration Guide*.

### Evaluator Classes

The evaluator classes contain methods that raise underwriting issues based upon the checking set. In the base configuration, the default evaluator class, DefaultUnderwriterEvaluator.gs, raises underwriting issues for all lines of business. Each line of business implements its own evaluator class. For example, the evaluator for the personal auto line of business is PA\_UnderwriterEvaluator.gs. These evaluator classes contain methods that raise underwriting issues for a checking set. For example, the onRenewal method in the default evaluator raises underwriting issues for the Renewal checking set.

If you make code changes to the methods that raise underwriting issues, the code must adhere to the following:

- The method must raise issues only for that checking set.
- Only one checking set can raise a given underwriting issue type.

In the default configuration, the evaluators always raise auto-approvable issues regardless of whether the current (or other) user has sufficient authority to approve them. The evaluators always raise these issues because the issue needs to exist for the least-privileged user of the system. This approach separates the underwriting guidelines for things require approval from the approval levels assigned to users. For users with sufficient authority to approve the resulting issue, PolicyCenter usually does not display the issue in the user interface. (The user can easily view the issue by selecting different view options.)

You can configure the application to raise auto-approvable issues only when one or more users would be unable to approve them. One reason to do this would be to improve performance by reducing the number of auto-approvable issues being raised and silently approved.

#### See also

- “Creating Underwriting Issues” on page 474

## Removing Orphaned Issues

Orphaned issues are underwriting issues that were generated at a checking set but are no longer an issue when that checking set runs at a later time. Therefore, issues that resulted from a previous version of the policy graph are removed if the data that caused them to be raised has been removed. For example, during a submission, PolicyCenter raised an issue for an expensive car on the policy. When the agent removes that car from the policy, the issue also disappears.

**Note:** PolicyCenter does not remove manual underwriting issues as orphans.

The `IPolicyEvaluationPlugin` removes orphaned issues after it runs the evaluators that create underwriting issues. The plugin removes or deactivates any issues for the current checking set that were not raised during that execution of the evaluators. Issues that currently have an approval or rejection associated with them are deactivated by setting the `Active` bit to `false`. Other issues are removed by setting the `ExpirationDate` to the slice date after which the conditions for the issue no longer exist. If a deactivated issue later reappears, the `Active` bit on the issue is set back to `true`, and any unexpired approval or rejection present on the issue applies again.

## Blocking Points

Blocking points represent points in the job at which an issue can block progress of the job.

Blocking points and checking sets are related, but not the same. In general, there is a checking set for each blocking point. However, there are other special checking sets as well as special blocking points. Issues need not have the same checking set and blocking point. For example, an issue can be created pre-quote, but not block until quote release or binding. The special blocking points are:

- **Non-Blocking** – An issue does not block progress.
- **Rejected** – A rejected issue blocks further progress because it prevents the job from crossing any blocking point.

In addition, there are blocking points for quote, quote release, bind, and issuance. The blocking points are described in the “Underwriting Issue Type System Table” on page 489.

The blocking point values have a priority order. When the job is at a blocking point, the job progress is blocked by any open issue that blocks at that priority or at a higher priority. (Generally, higher priority blocking points occur earlier in the job.) For example, a job has been quoted and is waiting to be bound. A loss-control specialist adds a referral reason to the policy that blocks quote. The resulting issue blocks binding the policy, even though the `Blocks Quote` blocking point has already been passed. The user can approve the issue then bind the policy without needing to obtain a new quote. In the default configuration, raising a lower priority blocking issue does not invalidate already completed steps.

The job process classes evaluate whether to raise underwriting issues and check for blocking issues in the `evaluateAndCheckForBlockingUWIssues` method. For example, the job process calls this method before binding a policy change. Before binding, the `PolicyChangeProcess` evaluates the all, pre-bind, and pre-issuance checking sets and then checks to see if any issues block issuance. PolicyCenter determines the current point at which an issue blocks both by its type and any associated approval or rejection. Thus, the issue blocks operation if the priority of the current point is greater than or equal to the priority of the operation. Therefore, a rejected issue blocks everything, a non-blocking issue blocks nothing. An issue that blocks bind also blocks issuance but does not block quoting.

### See also

- “Blocking Point” on page 491

## Configuring Underwriting Referral Reasons

Underwriting referral reasons are defined in the `UWIssueType` system table and have the `CheckingSet` column equal to `Referral`. In the base configuration, the policy evaluation plugin (`IPolicyEvaluationPlugin`) always checks for referral reasons on the policy, regardless of checking set or line of business. The `onReferral` method in the `DefaultUnderwriterEvaluator` class raises underwriting issues for referral reasons. This method creates a `UWIssue` based on that underwriting referral reason.

Referral reasons are generally added to a policy outside of a job. A PolicyCenter user can add referral reasons, for example, in response to an on-site visit or discussion with the insured. An external system can also add a referral reason. For example, an external claim system stores the loss claims for a policy. The external system notes that loss claims are unacceptably high. The external system sends a message to PolicyCenter to add a `UWReferralReason` on the policy. So the next time a job runs on the policy, the application creates an issue for this underwriting referral reason.

You cannot approve an `UWReferralReason`, but you can remove it. For example, a visit to the customer site revealed numerous safety violations. An underwriting referral reason was added to the policy. At subsequent visit, the safety violations had been addressed. The underwriting referral reason is removed from the policy.

Referral reasons are a convenient way to add fully-formed issues to a policy from an external application. These fully-formed issues do not require additional data from the policy in PolicyCenter. Using them in this way requires that the external application has knowledge of the underwriting guidelines, such as when to consider claim levels as excessive.

In some situations it may be more appropriate to avoid referral reasons, and use underwriting issues directly. You can configure PolicyCenter to make calls to an external application, and extend the `Policy` entity to store data from the external application. For example, a claim system does not know the total premium of the policy and cannot compute a loss ratio. So it cannot send a fully-formed referral reason to PolicyCenter. PolicyCenter can retrieve the loss information from the claim system when it evaluates the appropriate checking sets. (The default integration with a claim system implements this retrieval. See “Accessing Summary Loss Information from the Claim System” on page 736 in the *Application Guide*.) Alternately, the external claim system could use an API to send loss information to PolicyCenter. PolicyCenter could use this information when raising underwriting issues.

### See also

- “Working with Underwriting Referral Reasons” on page 420 in the *Application Guide*
- “Policy Web Services” on page 111 in the *Integration Guide*

## Job Interactions with Underwriting Issues

At specific places in the job flow, the job classes raise issues at checking sets and stop job progress because of blocking issues.

The following topics provide the checking and blocking points for each job type.

You can view and modify the Gosu code for each job. The Gosu code is in the `gw.job` package in Studio.

## Submission

The `SubmissionProcess` class handles checking sets and blocking points at the following times:

Submission job	Checking sets	Blocking points
Before quoting	All	BlocksQuote
	Referral	
	PreQuote	
	Question	
Before quote release	All	BlocksQuoteRelease
	Referral	
	PreQuoteRelease	
During bind, when not issuing	All	BlocksBind
	Referral	
	PreBind	
During bind, when also issuing	All	BlocksIssuance
	Referral	
	PreBind	
	PreIssuance	

## Renewal

Unlike other jobs, the `RenewalProcess` is often automated. As a result, renewal follows a different flow than the other jobs. There are a few primary differences in the automated flow:

- Checks are done and then repeated at various timeout points in the process.
- Blocking issues generate exceptions. When this occurs, the renewal is escalated to the underwriter.

The renewal job handles checking sets and blocking points at the following times:

Renewal job	Checking set	Blocking point
Before quoting	All	BlocksQuote
	Referral	
	PreQuote	
	Question	
Before quote release	All	BlocksQuoteRelease
	Referral	
	PreQuoteRelease	
Before scheduling the first check	All	BlocksIssuance
	Referral	
	Renewal	
	PreBind	
	PreIssuance	
During the first check	All	BlocksIssuance
	Referral	
	Renewal	
	PreBind	
	PreIssuance	

Renewal job	Checking set	Blocking point
During the final check	All	BlocksIssuance
	Referral	
	Renewal	
	PreBind	
	PreIssuance	
Before issuing the renewal	All	BlocksIssuance
	Referral	
	Renewal	
	PreBind	
	PreIssuance	

The code handles automated renewals differently from manual renewals. In automated renewals, all issues are run as *auto-approvable*. Issues are given automated approvals if the special renewal user has sufficient authority to progress the job. The `runMethodAsRenewalUser` method gets a user from the `getAutomatedRenewalUser` method in the `IPolicyRenewalPlugin`. In the default configuration, the user is `renewal_daemon`. The automated renewal runs the job as this user, using the authority profile of this user for approvals. This user approves the issues.

The `runMethodAsRenewalUser` method sets a flag on the `RenewalProcess` object indicating that the renewal is automated.

#### See also

- For more information about the `IPolicyRenewalPlugin`, see “Renewal Plugin” on page 165 in the *Integration Guide*.

#### Policy Change

The `PolicyChangeProcess` class handles checking sets and blocking points at the following times:

Policy change Job	Checking set	Blocking point
Before quoting	All	BlocksQuote
	Referral	
	PreQuote	
	Question	
Before quote release	All	BlocksQuoteRelease
	Referral	
	PreQuoteRelease	
During bind	All	BlocksIssuance
	Referral	
	PreBind	
	PreIssuance	

## Rewrite

The RewriteProcess class handles checking sets and blocking points at the following times:

Rewrite job	Checking set	Blocking point
Before quoting	All	BlocksQuote
	Referral	
	PreQuote	
	Question	
Before quote release	All	BlocksQuoteRelease
	Referral	
	PreQuoteRelease	
During bind	All	BlocksIssuance
	Referral	
	PreBind	
	PreIssuance	

## Issuance

The IssuanceProcess class handles checking sets and blocking points at the following times:

Issuance job	Checking sets	Blocking points
Before quoting	All	BlocksQuote
	Referral	
	PreQuote	
	Question	
Before quote release	All	BlocksQuoteRelease
	Referral	
	PreQuoteRelease	
During bind, when also issuing	All	BlocksBind
	Referral	
	PreBind	
	PreIssuance	

## Reinstatement

When you reinstate a canceled policy, approvals carry over from the canceled policy even if they were approved only until the next change. PolicyCenter does not consider reinstatement a change.

## Cancellation

Cancellation jobs do no checks for underwriting issues.

## Audit

Audit jobs do no checks for underwriting issues.

## Issue Keys

The `IssueType` and the `IssueKey` uniquely identify each `UWIssue`. For any given combination of issue type and issue key, there can only be one issue on a `PolicyPeriod` at any given point in time. The `PolicyEvalContext` automatically handles enforcement of this. When creating an issue, any pre-existing issue with the given type and key will be returned, and any deactivated issue that matches will be activated and then returned.

The key allows you to identify an issue as the same as another issue that might have been raised at another point in time. If you modify the code, and there happen to be two issues that are the same, then any approval or rejection applies to both of them. As a result, the method of forming the key for a given type of issue is critical. If the key matches unintentionally, approval or rejection of one issue incorrectly applies to the other. If the key does not match for issues that are the same, it can lead to the need to constantly reapprove a given issue.

Guidewire recommends that you do not change the algorithm for forming the key for an issue after moving that algorithm to production. If you change the algorithm, current issues may be treated as orphaned. PolicyCenter will create new copies of the issues, requiring new approvals or rejections.

### Best Practices

Guidewire recommends these best practices for forming issue keys.

For issues that affect the policy as a whole, such that there is only ever one such issue, use the code of the type as the key for the issue. Total policy premium is an example of this. Thus, if the premium goes up or down, the same issues will be edited, rather than raising new issues each time.

For issues that relate to a particular entity, have the issue key encode the `FixedID` of the entity, or potentially some other unique identifier such as the VIN number. For example, use this method for an issue relating to the collision deductible on a given vehicle. This method allows that same issue to be raised for several different vehicles. It also ensures that the same issue will be modified if changes affect that issue.

If the issue relates to a jurisdiction, create one issue per jurisdiction, with the jurisdiction encoded as part of the issue key. Since approvals for jurisdiction-based issues only apply to one jurisdiction at a time, rather than to a set of jurisdictions, each jurisdiction must have its own issue. An issue relating to a different jurisdiction would not reuse approvals or rejections. For example, an issue around the garage jurisdiction of a vehicle encodes both the `FixedID` or VIN of the vehicle as well as the jurisdiction. Changing the garage jurisdiction generates a new issue, rather than modifies the existing issue. This behavior also applies to set comparator types that you develop.

If the issue relates to a specific location, or a specific building at a certain location, encode the `FixedID` or other invariant identifier into the issue key. This encoding ensures that approving Building 2 in San Francisco to go without an alarm does not approve the same for Building 2 in Los Angeles.

## Configuring Underwriting Issue History

Underwriting history preserves an audit trail of issues and approvals, even on abandoned branches. To keep track of this history, every issue creation or removal and every issue approval generates an `UWIssueHistory` entity. The `Policy` entity has an `IssueHistories` array. This array contains the history of changes to all underwriting issues associated with the policy. The `UWIssueHistory` entity is non-effdated.

A `UWIssueHistory` entity is created automatically every time an issue is approved or rejected. Issue histories are also generated when the issue is reopened, removed, or when the approval expires. See the `gw.policy.UWIssueEnhancement.gsx` class for details.

An `UWIssueHistory` entry is also generated whenever an issue is either created or deactivated.

**Note:** In the default configuration, a history event is not created if values on an issue change. For example, an issue exists on a policy because there are six cars. Later the policy only has three cars. The checking set checks the current number of cars but does not generate a history event.

## UWIssueHistoryStatus Typelist

The Status field on the UWHistory entity is an UWIssueHistoryStatus typelist. The UWIssueHistoryStatus typelist represents the type of action recorded by the history. Values are:

Value	Description
Approved	The issue was approved.
Created	The issue was created.
Expired	The issue approval expired.
Rejected	The issue was rejected.
Removed	The issue was removed.
Reopened	The issue was reopened by removing approval or rejection.

## Comparing Issue Values

The values associated with UWIssues can be either numeric or non-numeric, as needed to describe level of risk or authority related to a specific issue type. The ValueComparator is used to determine if:

- The value of an issue is within the limits set on the existing approval
- The value of an issue is within the user's authority to approve
- A requested approval is within the user's authority to authorize

Additionally, the value itself is not strongly typed; its type is inferred from the ValueComparator assigned to the issue type.

The ValueComparator typelist contains the list of possible comparisons, such as *at least* and *at most*. The UWIssueValueComparatorWrapper class defines how to compare issue values. That class uses the UWIssueValueType class to define what the underlying value type is. The UWIssueValueType class defines how to convert the value that gets stored in the Value field on UWIssue or the ReferenceValue field on UWHistory. This class also defines how to validate raw string values. More than one comparator can use the same underlying value type.

The UWIssueValueType class has three methods: `deserialize`, `serialize`, and `validate`. This class defines two primary value types, represented as constants on the UWIssueValueType interface. The two value types are `BIG_DECIMAL` type, and `STATE_SET` type. The `BIG_DECIMAL` type converts and validates to and from a big decimal number. The `STATE_SET` is a comma-separated list of codes for jurisdictions. The `STATE_SET` type can either be an inclusive or exclusive set. If the list is preceded by *not*, a jurisdiction is part of the set if it is not in the list.

You can use `STATE_SET` as an example for writing comparators for other types of sets.

You can create comparators for other types of numeric values such as dates. Dates would need a greater than or equal comparator that operates on dates instead of numbers. The UWIssueValueComparatorWrapper is bound to a particular UWIssueValueType. The comparator wrapper implements a `compare(String, String)` method, which indicates whether or not the given issue value is within the bounds of the given grant value. The wrapper also implements a `calculateDefaultApprovalValue` method. The wrapper ties together an issue type to its underlying value type (if any). Every key in the ValueComparator typelist must have an associated subclass of UWIssueValueComparatorWrapper.

### See also

- “Comparator” on page 489

## Underwriting Issue Type System Table

You configure issue types in the `UWIssueType` system table in Product Designer. The `UWIssueType` system table displays `UWIssueType` entities.

**IMPORTANT** You must create and manage the set of issue types and their characteristics very carefully. Guidewire recommends that you do not remove issue types once they have been deployed in a production system. Guidewire also recommends that you do not change the `Code`, `Comparator`, `BlockingPoint`, `CheckingPoint`, and `AutoApprovable` fields after an issue is in production. It is permissible to alter the fields relating to approval defaults, along with the name or description of the issue.

PolicyCenter uses these default values in the following ways:

- As an initial setting in the user interface for approvals that do not already have an approval present
- As automated approvals for auto-approvable issues and issues in automated processes
- For determining availability of the **Approve** button, which indicates that you can approve a specific issue to the default level

**Note:** PolicyCenter creates approvals that use the default values for auto-approvable issues and automated processes, such as automatic renewal. All blocking points except `Non-blocking` can stop progress of the job pending manual approval. You need to carefully consider the impact of setting the default approval to block or not to block.

### Issues in the Underwriting Issue Type System Table

This topic describes the columns in the `UWIssueType` system table that relate to issues.

#### Code

Use the `Code` column to identify the issue type. The code must be unique for all issue types.

#### Name

Use the `Name` column to specify a name for the issue type. The name identifies the issue in the user interface.

#### Description

Use the `Description` column to specify a description for the issue type. The description appears in the user interface.

#### Comparator

Use the `Comparator` column to define how to compare the issue value. You can specify:

- If the value of an issue is within the authority granted to a user
- If value of an approval is within the authority granted to a user
- If the value of an issue is within the associated value of the approval

In the default configuration, the values are:

Value	Description
Any	For use only with authority grants. Signifies that the user has the authority to approve an issue of any value. You can specify this value to approve a value of any amount, or any value in a set.
At least Numeric_GE	The issue value must be greater than or equal to the approval or authority grant value. Treat the value as a <code>BigDecimal</code> . Select this value if the issue has an associated value where a smaller number is associated with more risk, such as a deductible.
At least Monetary_GE	The issue value must be greater than or equal to the approval or authority grant value. Treat the value as a <code>MonetaryAmount</code> . Select this value if the issue has an associated value where a smaller number is associated with more risk, such as a deductible.
At most Numeric_LE	The issue value must be less than or equal to the approval or authority grant value. Treat the value as a <code>BigDecimal</code> . Select this value if the issue has an associated value where a larger number is associated with more risk, such as total premiums or total insured value.
At most Monetary_LE	The issue value must be less than or equal to the approval or authority grant value. Treat the value as a <code>MonetaryAmount</code> . Select this value if the issue has an associated value where a larger number is associated with more risk, such as total premiums or total insured value.
In set State_Set	Treat the authority grant or approval value as a set of jurisdictions, and the issue value must be within that set.
None	Has no associated value. Use for issues that are either there or not.

The `ValueComparator` typelist defines these values. This typelist defines the comparators that issue values can use. You can add to the `ValueComparator` typelist. Each `ValueComparator` typekey has an associated `UWIssueValueComparatorWrapper` Gosu class that performs the comparison.

## Value Formatter

Use the `ValueFormatterType` column to define which output formatter to use. In the default configuration, the values are:

Value	Description
Age	Formats the value as an age followed by the abbreviation yrs.
Currency	Formats the value as the currency associated with the current locale.
Integer	Formats the value as an integer.
MonetaryAmount	Formats the value as a <code>MonetaryAmount</code> , which contains a value and a currency.
Number	Formats the value as a <code>BigDecimal</code> .
StateSet	Formats the value as a jurisdiction.
USD	Formats the value as a U.S. dollar (with cents).
USDBrief	Formats the value as a U.S. dollar (without cents).
Unformatted	Does not format the value.
Units	Formats the value as an integer followed by the word units.
TestFormatter	Used only for testing.

The `ValueFormatterType` typelist defines these values. The formatter types allow the presentation of values in a localizable and consistent manner in the user interface. Only output uses formatters. You can configure formatters to meet a wide range of output needs.

You can view the code for formatter types by navigating to `ValueFormatter.gs` located in the `gw.job.uw` package in Studio.

## Blocking Point

Use the `BlockingPoint` column to specify the point at which this issue blocks progress of a job. The values, listed in priority order from lowest to highest, are:

Value	Description
Non-Blocking	These are sometimes referred to as informational issues. These issues do not block progress unless specifically rejected by a user.
Blocks Issuance	Prevents issuance for submission jobs. Prevents binding for jobs that do not have a separate issuance step.
Blocks Bind	The issue blocks binding the policy.
Blocks Quote Release	Allows quoting to be performed, but unauthorized users cannot view the quote until an underwriter has the opportunity to review and approve it. The job cannot progress further.
Blocks Quote	Prevents quoting and steps leading up to quoting. Use <code>Blocks Quote</code> for issues that require intervention before the rating engine is called. <code>Blocks Quote</code> can be used for other types of issues, as well. However, if a job has a <code>Blocks Quote</code> issue, PolicyCenter will not call the rating engine, and the job will remain in draft state.  Do not use <code>Blocks Quote</code> for issues that need information from a valid quote. For example, do not use <code>Blocks Quote</code> for an issue related to total premium.
Rejected	The most restrictive value. Rejects the issue. A rejected issue prevents the job from crossing any blocking point.

The `UIssueBlockingPoint` typelist defines these values. This typelist represents the places at which an issue can block. The priority of the typecode is used to order the blocking point, with larger numbers indicating a more restrictive blocking point. The blocking point of an issue is based on the combination of the blocking point and approval, if present.

**Note:** Consider using the `Blocks quote release` blocking point rather than `Blocks Quote`. Some data, such as quote modifiers, are generated as a result of obtaining the quote. If this data triggers a `Blocks quote` issue, the user may have difficulty resolving the issue. The difficulty is because the user interface, as a result of not permitting quote, may no longer allow the user to edit those fields.

## Checking Set

Use the `CheckingSet` column to specify when to check for existence or absence of an issue type. The evaluator classes further enforce which jobs check for that issue type. In some cases, PolicyCenter uses the `CheckingSet` value to filter issue types for display in the user interface. Values are:

Value	Description
A11	PolicyCenter checks for this issue at all blocking points. An existing issue is removed if the evaluators do not trigger this type of issue. Issues in this checking set may come from external systems in a way that is not necessarily synchronized with the job flow. For example, a claim system sends notice of excessive claims, or a billing system sends notice of a delinquent account. If you have an issue that cannot change once the policy is quoted (without re-editing the policy), do not use this checking set.
Manual	The issue is created because the user manually added an issue to the policy period. PolicyCenter displays all issues with <code>Checking Set</code> equal to <code>Manual</code> on the <b>Risk Analysis</b> screen.
MVR	For motor vehicle record issues. PolicyCenter checks for this issue at quote, quote release, bind, and issuance.
PreBind	For issues that can only be detected immediately prior to binding the job. PolicyCenter checks for this issue prior to binding.
PreIssuance	For issues that can only be detected immediately prior to issuance. For submission jobs, PolicyCenter checks for this issue prior to issuing the policy. For jobs that do not have a separate issuance step, PolicyCenter checks for this issue prior to binding.

Value	Description
PreQuote	For issues that can be detected on the policy period even before quote, such as issues related to data entered in draft mode. PolicyCenter checks for this issue prior to generating a quote.
PreQuoteRelease	For issues that need information from a valid quote or that can be detected after a valid quote. For example, you can use this checking set for issues that depend on the premium. PolicyCenter checks for this issue after a valid quote, but before the quote is released.
Question	For issues is based on answers to a question set. PolicyCenter checks for this issue before quote. For information on how to associate an issue with a question in Studio, see "Defining New Questions" on page 62 in the <i>Product Model Guide</i> .
Referral	For issues is related to underwriting referral reasons on the policy. PolicyCenter checks for this issue at all blocking points. PolicyCenter displays all issues with Checking Set equal to Referral on screens where users can add a UWReferralReason to the policy. You can also add a referral reason to a policy by using the addReferralReason API.
RegulatoryHold	For issues related to policy holds with the <b>Regulatory Hold</b> hold type. PolicyCenter checks for this issue at all blocking points. Therefore, the user is notified each time the job advances to the next step.
Reinsurance	For issues related to reinsurance. For example, you can use this checking set to raise an issue if a policy has insufficient reinsurance. PolicyCenter checks for this issue at all blocking points except quote.
Renewal	For issues related to renewals. PolicyCenter checks for this issue in renewal jobs at quote and issuance.
Upgrade	In certain cases, Guidewire uses this value when upgrading PolicyCenter. Do not use this value for any other purpose.
UWHold	Use for issues related to policy holds with the <b>Underwriting Hold</b> hold type. PolicyCenter checks for this issue at all blocking points. Therefore, the user is notified each time the job advances to the next step.

The `UWIssueCheckingSet` typelist defines these values. The checking set represents the place in a job that an issue is checked for. An issue is checked for at only one checking set.

#### See also

- “[Checking Sets and Evaluators](#)” on page 481
- “[Evaluator Classes](#)” on page 481

#### Auto-approvable

Use the `AutoApprovable` column to specify whether or not this issue is treated as a auto-approvable issue. Auto-approvable issues can be automatically approved if the user has an authority grant that permits an approval at the default level. If PolicyCenter can approve all blocking auto-approvable issues at the default level, then PolicyCenter approves all blocking issues. Otherwise, PolicyCenter approves none. The default level is the amount in the authority grant plus any offset amount or percentage from the `UWIssueType` system table.

Set `AutoApprovable` to `false` for issues which are meant to be explicitly approved by an authorized user. Typically, these are the judgement calls that the underwriter makes. If an issue corresponds to something unusual, that requires review before approving, then set `AutoApprovable` to `false`.

Set `AutoApprovable` to `true` for issues that, while they prevent unauthorized users from progressing the job, they are not unusual if the user is authorized. Automatically approving common issues allows the user to focus on the more important and unusual issues. For example, you may want to automatically approve issues raised to confirm the jurisdictions in which an agent is licensed or authorized to write business. Thresholds on sizes or types of policies or changes might also be automatically approved.

Values are `true` or `false`.

**See also**

- “Configuring Authority Grants” on page 478

## Default Approval Values in Underwriting Issue Type System Table

This topic describes the columns in the UWIssueType system table that relate to approving issues. These columns specify default approval values.

### Default Edit Before Bind

Use the DefaultEditBeforeBind column to specify the default value of the EditBeforeBind property on new approvals. If `false`, the approval is removed if the policy is edited before binding this job. If `true`, the approval remains if the policy is edited before binding.

### Default Duration Type

Use the DefaultDurationType column to specify the default value of the DurationType property on new approvals. Use this property to indicate when an approval will expire. The UWApprovalDurationType typelist defines these values. Values are:

Value	Description
NextChange	The approval will expire on the next issuance, policy change, renewal, or rewrite job.
EndOfTerm	The approval will expire on the next renewal or rewrite job.
OneYear	The approval will expire one year (minus one day) from the current EditEffectiveDate of this job. PolicyCenter calculates the expiration date then stores it in the ApprovalInvalidFrom field on the UWIssueApproval object.
ThreeYears	The approval expires three years (minus one day) from the current EditEffectiveDate of this job. PolicyCenter calculates the expiration date then stores it in the ApprovalInvalidFrom field on the UWIssueApproval object.
Rescinded	Valid until the approval is rescinded. The approval does not expire on its own.

When determining the validity of the approval, the application looks at the effective date of the current job and of future jobs. For example, suppose you have a one year approval at the beginning of a six month policy. This approval extends through any policy changes in the current term, the six month renewal, and any policy changes in the following term. It does not cover a second renewal.

### Default Approval Blocking Point

Use the DefaultApprovalBlockingPoint column to specify the default approval blocking point. This value must be at least one level higher than the BlockingPoint, or both must be Non-Blocking. This column has the same values as the BlockingPoint column.

### Default Value Assignment Type

Use the DefaultValueAssignmentType column to specify how to compute a default approval value from the value of the issue. PolicyCenter uses this value if Comparator is At least or At most. The UWValueAssignmentType typelist defines these values. Values are:

Value	Description
<code>null</code>	Set this value to null for issues that have a non-numeric value or that do not have an associated value.
Fixed	The default approval value is copied directly from the issue value.

Value	Description
Offset Amount	The DefaultValueOffsetAmount is added to (or subtracted from, depending upon the implementation of the Comparator) the issue value to produce the default approval value. Use this setting only with issues that have a numeric value.
Offset Percentage	The DefaultValueOffsetAmount is treated as a percentage increase or decrease, and the issue value is offset by that percentage to produce the default approval value. Use this setting only with issues that have a numeric value.

### Default Value Offset Amount

Use the DefaultValueOffsetAmount column if DefaultValueAssignmentType is Offset Amount or Offset Percentage. If the DefaultValueAssignmentType is set to OffsetAmount or OffsetPercent, this field is the associated value or percent to use in the offset calculation. Positive values of DefaultValueOffsetAmount will produce a default approval value that is somewhat riskier than the issue value.

### Default Value Offset Currency

Use the DefaultValueOffsetCurrency column if DefaultValueAssignmentType is Offset Amount. This field sets the default currency to use in the offset calculation.

## Additional Information About the Underwriting Issue Type System Table

This topic contains additional information about the Underwriting Issue Type system table.

### Columns for Assignment Type and Value Offset Amount Default Values

The DefaultValueAssignmentType column is used if the comparator is At least or At most. In turn, if DefaultValueAssignmentType is OffsetAmount, the DefaultValueOffsetAmount and DefaultValueOffsetCurrency are used. You can extend this default setting behavior for other numeric types. Positive offsets reflect increased underwriting risk for the at least and at most comparators. For example, the at most comparator can be used for determining the risk associated with the value of a car. If the authority grant provides default approval at \$100,000 with 10% offset, the user can approve a car whose value is as high as \$110,000. The *at least* comparator can be used for determining the risk of a deductible. If the authority grant provides default approval for at least \$1000, with 10% offset, then the user can approve a deductible as low as \$900.

### Underwriter Issue Type Verifier Class

The UWIssueTypeVerifier class enforces constraints on the issue type definition in the UWIssueType system table to prevent certain types of configuration errors. This class is in the gw.systables.verifier package in Studio.

In the default configuration, this class enforces the following:

- Either the DefaultApprovalBlockingPoint must be one level higher than the issue BlockingPoint, or both must be non-blocking.
- The DefaultValueAssignmentType must be defined if the Comparator is At least or At most. The codes for these comparators are Numeric\_LE and Numeric\_GE.
- The DefaultValueOffsetAmount must be set if the assignment type is Offset Amount or Offset Percentage. The codes for these values are OffsetAmount and OffsetPercent.

### See also

- “Underwriting Issue Type System Table” on page 489
- The UWIssueType entity in “Underwriting Issue Object Model” on page 480

## Configuring Approvals

PolicyCenter creates an approval when you approve or reject an issue. The `UWIssueApproval` object represents an approval. To approve an issue, you must have an authority grant in an authority profile for that issue type. If the issue has a value, the authority grant must be valid for that value.

**Note:** In the default configuration, the **Approve** button is only available when the user has the authority to approve the issue to its default level. You can modify the configuration if you have a different use case.

Approvals can be generated either by a user clicking to approve or reject an issue, or automatically by the application. Automatic approval is most commonly seen for auto-approvable issues, but also occurs during unattended processing, such as in routine renewal. You can identify automatically-generated approval through the `IsManual` property.

At any point in effective time, each `UWIssue` can have at most one approval. If you reapprove an issue, PolicyCenter removes any existing approvals from that point forward in effective time. Approvals always extend from the current slice the user is working with until the end of the period. An issue is eligible for approval if it currently blocks progress and has not been rejected. If the issue has already been rejected, it must be reopened before being approved. If the issue has been approved through issuance with a sufficient value, it must be reopened before being approved for a different value or different approval settings.

### Rejecting an Issue

Rejections generate an approval, but with the `BlockingPoint` set to `Rejected`. A rejection prevents the job from crossing any blocking point. The user must have the `uwreject` permission. The user can reject an issue if:

- It is eligible for approval.
- It is a non-blocking issue and has not already been rejected.

**Note:** The user need not have sufficient authority to approve the issue to reject it.

### Reopening an Issue

Reopening an issue removes an approval from the issue. The issue then blocks at the blocking point associated with the issue type. To reopen an issue, you must have the `uwreopen` permission or be able to approve the issue.

### Approvals for Auto-approvable Issues

Issues defined as auto-approvable are given automated approvals during issue evaluation, provided that all of the following are true:

- The issue does not already have an approval or rejection.
- The approval will unblock progress.
- No non-auto-approvable issues block progress.
- No other issues block progress.
- The user has the authority to approve all blocking, auto-approvable issues to their default values.

If the preceding are true, then PolicyCenter automatically, and transparently, makes that set of approvals as the user progresses the job.

---

**IMPORTANT** Guidewire recommends that you define all auto-approvable issues with `DefaultValueAssignmentType` set to `Fixed`. This setting avoids a situation where the user can approve the issue, but the default approval is outside of their authority.

---

An automated approval has an `AutomaticApprovalCause` which you can use to identify these approvals in the user interface. In the default configuration, this value is `jobNumber@BlockingPoint` for automated approvals that happen during normal job processes.

**Note:** During automated processing, such as unattended renewals, the auto-approvable approval logic applies to all issues, including those that are not identified as auto-approvable.

## Approval Expiration

Approvals can be set to expire after a certain point in time by specifying one or three years in the `DefaultDurationType` column of the `UWIssueType` system table. When an approval of this type is created, the expiration date is calculated and written to the `ApprovalInvalidFrom` field.

## Special Approval Permission

There is a special approval permission that allows a user to approve an issue for any value. This permission is `uwapproveall`. Guidewire provides this permission to approve issues that no user can approve. Guidewire recommends that you never use this permission unless there is a time-critical policy that cannot be advanced by any underwriter. An incomplete or misconfigured set of authority profiles can cause this situation. In the default configuration, only `su` is given this permission.

A user with the `uwapproveall` permission is presented with a **Special Approve** button that allows them to approve any issue that they could not otherwise approve. The user must confirm that they wish to use this permission before the application displays the screen that allows them to approve the issue.

## Passing Approval Requests to Underwriters

In the underwriting approval process, the policy period can be referred to users with higher or different authority profiles. These users receive activities notifying them they need to approve issues on the policy. A particular policy may be referred to several users before being approved. When the underwriter feels that the review of the policy is complete, whether issues have been approved or not, the underwriter clicks the **Release Lock** button. This action releases the policy from underwriting, and PolicyCenter sends an activity to the user who initiated the referral. The `InitialReferrer` role stores the name of this user. (The underwriter can modify the recipient and other fields in the activity.)

To facilitate passing policies between agents and underwriters, a given policy period can be flagged as `edit locked` and `quote hidden`. After an agent hands these policies to an underwriter, the agent usually cannot edit the policy or view the quote. The permissions for editing or viewing the quote while under review are usually reserved for underwriters.

### Edit Locked Flag

When the `EditLocked` flag is `true`, only users with the `editlockoverride` permission can edit the policy. The intention is that this permission is granted to underwriters but not to producers. The `EditLocked` flag is set to `true` when the user clicks the **Request Approval** button to request approval from another user. If a policy has not been formally submitted for underwriting review, the underwriter expects the quote to be hidden and the policy locked for editing. Therefore, the `EditLocked` flag is also set to `true` whenever a user with the `editlockoverride` permission quotes a policy. The `EditLocked` flag is set back to `false` when the underwriter sends the policy back to an agent by clicking the **Release** button on the associated activity. (Click **Release Lock** to get to the activity.) The **Release** button is available even if the policy has blocking issues. The `EditLocked` flag is also set to `false` when the policy is bound and/or issued. The `EditLocked` flag is always set to `false` when a job starts.

**Note:** The PCF files check the `EditLocked` flag to determine whether or not the policy can be edited. The `EditLocked` flag does not lock the `PolicyPeriod` object.

## Quote Hidden Flag

When the `QuoteHidden` flag is set to `true`, only users with the `quotehideoverride` permission can see the quote for that `PolicyPeriod`. Similar to the `editlockoverride` permission, the intention is that the permission only be granted to underwriters.

The `QuoteHidden` flag is set to `true` when requesting approval if the job does not already have a valid quote. The flag is also set to `true` during quoting if the `PreQuoteRelease` blocking point has blocking issues or if the policy being quoted is marked as `EditLocked`. Thus, if an underwriter with `editlockoverride` quotes a policy, the policy will end up both edit locked and quote hidden. The `QuoteHidden` flag is set to `false` if a user, such as a producer, who does not have the `editlockoverride` permission approves all issues blocking quote release.

In the default configuration, when `EditLocked` is set to `false`, `QuoteHidden` is also set to `false`. Invalidating the quote also sets `QuoteHidden` to `false`. However, for policies that are under edit lock, `QuoteHidden` is immediately set to `true` on quoting. Thus any policy in underwriting review will have its quote hidden if it has ever had an invalid quote while under review.

**Note:** The PCF files for the job wizards determine whether to display the quote step by calling the `canViewQuote` method in the `JobProcess` class. The `canViewQuote` method checks the value of the `QuoteHidden` flag. The `QuoteHidden` flag does not lock the `PolicyPeriod` object.

## Initial Referrer User Role

The `InitialReferrer` user role facilitates passing policies between underwriting and producers. The initial referrer is the user who referred the policy to underwriting. When requesting approval for issues on a policy period, if the `InitialReferrer` user role is not set, the role is set to the current user. When the underwriter releases the policy by clicking `Release`, the `InitialReferrer` role is unset. An activity is sent to the `InitialReferrer` on the period. You can customize this by using activity assignment as described in “Assignment in PolicyCenter” on page 53 in the *Rules Guide*. You can also customize the `InitialReferrer` user role.

The `InitialReferrer` user role is a transient role assigned to users in the context of their role in the job. Because it is a transient role, it is assigned by the job code rather than being assigned to the user in the `Administration` tab. Because it is a transient role, it is not in the `UserRole` typelist.

# Handling Underwriting Issues in Policy Revisions

This topic describes how PolicyCenter handles underwriting issues in policy revisions. Policy revisions reflect changes to the policy over a period of time.

## Handling Underwriting Issues in Out-of-sequence Policy Changes

This topic describes how PolicyCenter handles underwriting issues in out-of-sequence policy changes. For detailed information on how PolicyCenter handles out-of-sequence changes, see “Out-of-sequence Jobs” on page 505 in the *Application Guide*.

PolicyCenter handles out-of-sequence policy changes by evaluating every future slice in the policy when it evaluates whether to raise underwriting issues. PolicyCenter first evaluates the slice. Then PolicyCenter checks to see if the slice is blocked and automatically approves auto-approvable issues. PolicyCenter evaluates whether to raise underwriting issues on the slice. The data can change such that the value of an issue changes in the future, or the issue can disappear and reappear in different slices. When this occurs, the issue is automatically split and removed as necessary. The `PolicyEvalContext` object pulls issues forward or backwards in time as necessary during evaluation. However, any issue with the same type and key will have the same `FixedID` if it is not completely removed from the policy as of the period start date.

Approvals are effective as of the slice date on which they are approved until the end of the period. When an issue is reopened or reapproved as of a particular effective date, PolicyCenter removes all approvals for that issue from that date forward. The behavior is the same for reopened and reapproved issues because a reapproved issue is automatically reopened. Thus, an approval at time T1 will extend through T2 until the end of the period, and any approval already at T2 will be removed. If the approval given at T1 is insufficient for time T2, the user will need to reopen the issue at T1 and create a more lenient approval. Alternately, the user can move T2 in effective time and reapprove the issue at time T2. PolicyCenter expires the original approval at T2 and creates a new approval from T2 until the end of the period.

If underwriting issues block future slices, PolicyCenter uses the same slice selector that it uses for out-of-sequence validation failures. PolicyCenter displays the selector if the `FailedOOSEvaluation` flag is set to true on the period. This flag indicates that the policy period is out-of-sequence. The flag is set if a future slice has blocking issues but the current slice is not blocking. The flag can also be set if any issues that are generated have values that vary over time (which includes the issue itself disappearing and reappearing).

In the default configuration, PolicyCenter adds a section sign, §, to indicate that the issue varies in time. PolicyCenter displays the section sign if the `ValueVariesAcrossSlices` and `IssueBlocksAtSomeSlice` properties in `UWIssueEnhancement` are true. PolicyCenter displays the section sign in parentheses, (§), if the value varies in time but is not currently blocking any slice. Lastly, the user interface displays a **Next Blocked Date** button that automatically takes the user to the next slice in effective time where issues are blocked. The **Next Blocked Date** button appears if there are blocking issues in future slices. The user can step through the slices, approving issues from that point forward until the policy is no longer blocked. To see how PolicyCenter displays this in the user interface, see “UW Issues on the Risk Analysis Screen” on page 418 in the *Application Guide*.

## Handling Underwriting Issues in Preempted Jobs

This topic describes how PolicyCenter handles underwriting issues in preempted jobs.

A preemption occurs if one job is in progress when another job on the same policy binds. For example, a job starts on a policy and creates policy period PP1, which is not yet bound. Then another job on the same policy binds, resulting in a bound policy period PP2 that preempts PP1. When the user handles the preemption on PP1, PolicyCenter creates a new policy period PP3. PolicyCenter handles underwriting issues as follows:

- All issues and approvals which are present in PP2 are carried over to PP3. Approvals from PP2 are not expired. Therefore, if PP2 includes an approval which is invalid from the next change, that approval remains in PP3.
- If PP1 includes a new issue which does not appear in PP2, that new issue does not appear in PP3.
- If PP1 includes changes to an issue or approval that is present in PP2, the issue appears in PP3 along with the approval from PP2. However, the issue history includes links to an invalid job. (The links still refer to PP1, which was preempted and is no longer valid.)

### See also

- “Preempted Jobs” on page 507 in the *Application Guide*

# Configuring the Account Holder Info Screen

The **Account Holder Info** screen is an example of a summary screen which consolidates information about an account holder from PolicyCenter and other applications. The **Account Holder Info** screen brings together account holder profile information from PolicyCenter, BillingCenter, ClaimCenter, and ContactManager. You can create similar screens which display summary and status information for an individual contact, based on selected account contact roles held by that contact. This summary screen can gather this information from Guidewire PolicyCenter, Guidewire BillingCenter, Guidewire ClaimCenter, and Guidewire ContactManager. You can configure this screen to obtain information from other third-party applications.

In the default configuration, some fields in the **Account Holder Info** screen display meaningful data only if PolicyCenter is integrated with BillingCenter, ClaimCenter, and ContactManager. Through configuration, you can retrieve equivalent information from third-party billing, claims, and contact systems of record.

The **Account Holder Info** screen includes information such as:

- Basic information, including the account holder's name, number of accounts on which this contact is the account holder, and in-force premium value across all policies on those accounts.
- Billing summary, including total billed and unbilled, total past due, and total outstanding.
- Value metrics, including first policy effective year, number of active policies, number and type of cancellations, lifetime premium value, open claim count, and net total incurred on those claims.
- Various alerts, such as past-due payments and open claims.
- Summary of work orders in progress, with links to work order details.
- Summary of open claims, with links to the affected policies.

This topic provides configuration information about the **Account Holder Info** screen and guidance for creating similar screens.

This topic includes:

- “Ways to Configure the Account Holder Info Screen” on page 500
- “Configuration Files for the Account Holder Info Screen” on page 503

**See also**

- “Account Holder Information” on page 375 in the *Application Guide*
- “Viewing the Account Holder Info Screen” on page 384 in the *Application Guide*

## Ways to Configure the Account Holder Info Screen

The **Account Holder Info** screen displays a summary of account holder data retrieved from PolicyCenter and other applications. You can configure the **Account Holder Info** screen to meet your needs. You can also use the **Account Holder Info** screen as a starting point for creating your own summary screen. If you create a screen based on the **Account Holder Info** screen, Guidewire suggests that you include the features described in this topic.

The **Account Holder Info** screen retrieves information from other InsuranceSuite applications of the same release version. With additional configuration, the screen can work with other versions of these products, as well as with third-party billing, claims, and contact management systems.

### Consolidate Account Information

You can modify the **Account Holder Info** screen in the default configuration to meet additional requirements.

The **Account Holder Info** screen is implemented in the `ContactFile_AccountHolder` PCF file. You can view and edit this PCF file in Studio by navigating to `configuration → config → Page Configuration → pcf → contactfile`. The `canVisit` property on this screen determines access to this screen and its link in the left sidebar. The screen and its link are visible only when the selected contact has a **Contact Role** of **Account Holder** on at least one account. You can extend this configuration to display the screen for other roles.

To configure the **Account Holder Info** screen to appear when other contact roles are selected, change the screen’s `canVisit` property.

#### To add a secondary account holder contact role

You can add a Secondary Account Holder contact role and display the **Account Holder Info** screen when you select a contact with this role. To do this:

1. Create an appropriate new contact role if needed; for example, `SecondaryAcctHolder_Ext`, then restart the server.

**Note:** The only built-in role for which the **Account Holder Info** screen displays meaningful, accurate data is the Account Holder contact role. You can configure any contact role to display the **Account Holder Info** screen. However, the screen displays data that reflects the value and responsibility of the contact. In the default implementation, the data focuses on account ownership. For example, billing values summarize financial responsibility of the contact and premium metrics reflect the monetary value the contact has with the carrier. These values only make sense if the contact owns an account or has a significant ownership role for the all of the policies on an account. If you enable the **Account Holder Info** screen for non-ownership contact roles, the values it displays are misleading.

For example, the **Account Holder Info** screen displays meaningful data for a second account holder contact role (for example, a spouse) that is on a policy. In this case, the spouse has significant ownership of the policies on the account. The data for the second account holder is similar to the data for the primary account holder.

On the other hand, the **Account Holder Info** screen may display misleading, inaccurate data for a contact role such as Named Insured. For example, an account includes a policy owned by a child. Displaying the **Account Holder Info** screen for that named insured displays data for all policies on the account, not just the one owned by that named insured. Therefore the screen does not reflect the overall value of the contact. For this reason, do not enable the **Account Holder Info** screen for contact roles such as Named Insured.

2. In Studio, open the `ContactFile_AccountHolder` PCF file.
3. Select the `Page` element to display its properties in the workspace area at the bottom of the screen.

4. On the **Properties** tab, change the `canVisit` property to include the name of the new contact role. The default `canVisit` property displays the screen if the user has `viewaccountholderinfo` permissions and the selected contact is an account holder of more than zero accounts:

```
perm.System.viewaccountholderinfo and
(new gw.web.contact.AccountHolderPolicyMetrics().countAccountsFor(contact, {AccountHolder}) > 0)
```

To keep the default behavior and display the **Account Holder Info** screen for the new contact role, you can add the new contact to the array of roles:

```
perm.System.viewaccountholderinfo and
(new gw.web.contact.AccountHolderPolicyMetrics().countAccountsFor(contact,
{AccountHolder, SecondaryAcctHolder}) > 0)
```

5. On the **Variables** tab, select the `roles` variable. Change its `initialValue` to include the new role. For example:  
`{AccountHolder, SecondaryAcctHolder}`

## Cross-application Alerts in a Single Location

On the **Account Holder Info** screen, you can view the alerts pertaining to the selected account holder. The default implementation includes a framework for creating and displaying alerts.

The **Account Holder Info** screen displays alerts in an `InputColumn` of the `AccountHolderDV` detail view. The `addAlert` method in the `gw.web.contact.ContactMetricsImpl` class adds alerts to this `InputColumn`. This class contains two methods to create new alerts: `createPolicyAlerts` and `createClaimAlerts`. Each of these methods defines one alert:

- `createPolicyAlerts` – Displays an alert if any cancellations are in progress on any policy belonging to the selected contact.
- `createClaimAlerts` – Displays an alert if there are any open claims on any policy belonging to the selected contact.

During implementation, you can add other alerts by modifying these methods.

**Note:** When extending the implementation, you can either modify the code directly in its class, or make the needed changes in a subclass of the original class. Guidewire recommends extending the implementation in a subclass to insulate your code modifications from future product updates.

Another type of alert is a label that the **Account Holder Info** screen displays when certain conditions exist. This behavior is controlled by the `visible` property of the label. An example of this type of alert is the `NotDirectBillAlert` label at the bottom of the billing summary `InputColumn`. This label is visible only if the **Billing Method** is not **Direct Bill**. You can add additional alerts of this type by adding labels to the screen and configuring their `Visible` properties.

## Links to Referenced Objects

When the **Account Holder Info** screen displays information about an account, work order, or claim, the screen provides a link to the referenced item.

The **Account Holder Info** screen provides links that take you directly to the referenced item. Links enable you to jump directly to:

- **Accounts** – The number of accounts listed in the upper left portion of the screen links directly to the **Contact File Accounts** screen. The **Contact File Accounts** screen contains additional links for each account the contact holds. Each of those additional links takes the user to the **Account File Summary** screen for that account.
- **Work Orders** – The **Work Orders in Progress** section lists each of the work orders that are currently in progress on the accounts on which the contact is the account holder. This section also lists the work order creation date, policy number, work order number, type, and status. Links enable you to jump directly to the **Policy Summary** screen for the policy to which the work order applies, and to the applicable open work order details screen.

- **Claims** – The **Open Claims** section lists each open claim associated with the selected account holder. This section also lists the insurance product, named insured (which may be different than the account holder), loss date, claim number, status, and total incurred amount. Links enable you to jump directly to the **Policy Summary** screen for the policy in which the claim has been made. From there, you can easily jump to the account to which the policy belongs. Then select **Claims** to view claim details, and jump directly to the claim in the claim system, assuming you have the required permissions.

## Controlled Access to the Account Holder Info Screen

As a first level of access control, administrators grant or revoke access to the **Account Holder Info** screen or summary screen that you implement. Access is based on user role. The screen must not provide access to data to which the user otherwise would not have access. For example, if you have access to the screen but do not have permission to view billing data, the screen must not display billing data.

The **Account Holder Info** screen uses the **Account Holder Info** (`viewaccountholderinfo`) permission to control access. You must add this permission to the roles for users who will view the **Account Holder Info** screen. Users who have permission to navigate to and view the **Account Holder Info** screen need additional permissions to see some of the summary information:

- **View Claim System** (`viewclaimsystem`) permission is required to view summary claim information.
- **View Billing System** (`viewbillingsystem`) permission is required to view billing summary information.

The **Account Holder Info** screen also respects producer code security. If you do not have permission to view a particular account or policy, the **Account Holder Info** screen does not display that information, even though the screen itself is available. Furthermore, any totals that the **Account Holder Info** screen displays apply only to the accounts and policies for which you have permission to view.

## Customizing the Screen Data

The **Account Holder Info** screen provides access to the underlying calculations and data. You can add or remove data items.

For example, you can make the following types of changes:

- Display summary information for additional contact roles, such as a **Secondary Account Holder**.
- Add and remove the individual data items that the screen displays.
- Change the Gosu code that displays the summary information.

## Changing the Length of Time for Cancellations

You can change the length of time (going backwards from today) for which to display information about cancellations. In Studio, navigate to the `gw.web.contact.AccountHolderPolicyMetrics` Gosu class. Modify the constant `CANCELLATION_MONTHS_SINCE` from its default value of 12 to a different value. The `cancellationsResultFor` method uses this constant.

## Changing the Formula that Calculates Lifetime Premium

You can change the formula that calculates **Lifetime Premium**. In Studio, navigate to the `gw.web.contact.AccountHolderPolicyMetrics` Gosu class. Modify the code in the `calculateLifetimePremium` method.

## Performance

Retrieving data for the **Account Holder Info** screen may take additional time when compared with the performance of other screens. Ensure that the performance of this screen does not impact the performance of routine operations that do not involve this screen.

For example, displaying the information requested by the **Account Holder Info** screen may be time-consuming. To avoid a negative impact on other PolicyCenter operations, the **Account Holder Info** screen is not the default screen on the **Contact** tab. Instead, when navigating to the **Contact** location, PolicyCenter displays the **Summary** screen. You must click a link in the left sidebar to view the **Account Holder Info** screen.

## Configuration Files for the Account Holder Info Screen

The following topics describe the configuration files that the **Account Holder Info** screen uses.

### Entities and the Account Holder Info Screen

On the **Policy** entity, the **OriginalEffectiveDate** property stores the date on which the policy was originally issued or bound. This property enables the display of the **First Policy Effective Year**. The **PriorPremiums\_amt** property stores premiums for policy terms prior to converting from a legacy system to PolicyCenter. This property enables the display of the **Lifetime Premium**. The system can set this value during conversion on renewal.

### Permissions for the Account Holder Info Screen

In Studio, the **configuration → config → Extensions → Typelist** folder contains the following permission:

- **SystemPermissionType.ttx** – Permission for **viewaccountholderinfo**

### Display Keys for the Account Holder Info Screen

In Studio, the **configuration → config → Localizations** folder contains files that define properties for the **Account Holder Info** screen.

- **display.properties** – Contains display key values for the **Account Holder Info** screen.

### PCF Files for the Account Holder Info Screen

In Studio, the **configuration → config → Page Configuration → pcf → contactfile** folder contains files that the **Account Holder Info** screen uses.

- **ContactFile.pcf** – Location group (menu) that adds the **Account Holder Info** link.
- **ContactFile\_AccountHolder.pcf** – The main **Account Holder Info** page container.
- **ContactFile\_WorkOrdersLV.pcf** – The **Work Orders in Progress** list near the bottom of the **Account Holder Info** screen.
- **ContactClaimsLV.pcf** – The **Open Claims** list at the bottom of the **Account Holder Info** screen.

### Gosu Files for the Account Holder Info Screen

In Studio, the **gw.web.contact** package contains Gosu files that the **Account Holder Info** screen uses.

- **AccountHolderBillingMetrics.gs** – Helper class that provides access to billing account information to calculate account holder billing metrics.
- **AccountHolderClaimMetrics.gs** – Helper class that provides access to claims for policies belonging to account holders.
- **AccountHolderPolicyMetrics.gs** – Helper class that provides access to metrics associated with policies belonging to account holders.
- **ContactMetrics.gs** – Collects metrics for accounts associated with contacts through a specified **AccountContact** role.
- **ContactMetricsFactory.gs** – Creates new instances of **ContactMetrics** objects each time a contact with the specified **AccountContact** role is selected.

- `ContactMetricsImpl.gs` – Collects the policy metrics for the accounts associated with a contact through a specified `AccountContact` role.

### Gosu Class that Provides Data to the Account Holder Info Screen

Variables and methods defined in the `gw.web.contact.ContactMetricsImpl` Gosu class provide data for the **Account Holder Info** screen. The helper class `gw.web.contact.AccountHolderPolicyMetrics` provides access to various policy metrics for the selected account holder contact.

**Note:** If you extend the functionality of the **Account Holder Info** screen, you can modify the code directly in its class, or make the needed changes in a subclass. Guidewire recommends that you extend the implementation in a subclass to insulate your code modifications from future product updates.

# Configuring Policy Data Spreadsheet Import/Export

In the default configuration, policy data import/export is provided in the commercial property line of business for buildings and locations. Policy data is imported and exported to a spreadsheet in .xlsx format. You can add policy data import/export of buildings and locations to other lines of business. You can also implement policy data import/export for other entity types, such as vehicles in a commercial auto policy.

This topic includes:

- “Changing the Spreadsheet Protection Password” on page 505
- “Configuring Spreadsheet Import/Export in Commercial Property” on page 506
- “Adding Spreadsheet Import/Export to Other Entities” on page 506

**See also**

- “Policy Data Spreadsheet Import/Export” on page 467 in the *Application Guide*
- “Importing and Exporting Policy Data Spreadsheets” on page 712 in the *Application Guide*

## Changing the Spreadsheet Protection Password

The worksheets within the exported spreadsheets are protected and designed to be used while protected. In the default configuration, the password is set to 1234.

The spreadsheet protection password is not intended to provide security. It exists to help spreadsheet users maintain the integrity of the data in the spreadsheet. If the data in your exported spreadsheets is sensitive, you must implement appropriate measures to ensure the required level of security.

**To change the worksheet protection password**

1. In Studio, navigate to `gw.excelimport` and open `ExcelExporter.gs`.

2. In the ExcelExporter class, find the SPREADSHEET\_PASSWORD variable assignment:

```
class ExcelExporter {
 // password to unprotect spreadsheet
 static final var SPREADSHEET_PASSWORD = "1234"
```

3. Change the value of SPREADSHEET\_PASSWORD.  
4. Restart PolicyCenter to deploy the change.

## Configuring Spreadsheet Import/Export in Commercial Property

In commercial property, policy data spreadsheet import/export is accessible on the **Buildings and Locations** screen, CPBuildingsScreen.pcf. This PCF file contains controls that initiate import and export.

## Adding Spreadsheet Import/Export to Other Entities

In the default configuration, policy data import/export is provided in the commercial property line of business for buildings and locations. You can add policy data import/export to other entities, including entities in other lines of business, by following these steps:

- “Step 1: Create an XML File Describing Columns to Import and Export” on page 506
- “Step 2: Define Column Data Resolvers” on page 508
- “Step 3: Create an Import Strategy for Each New Data Column Resolver” on page 509
- “Step 4: Modify the ImportStrategyRegistry Class” on page 510

### Step 1: Create an XML File Describing Columns to Import and Export

For each entity that you want to export and import, create an XML file that describes the columns to export and import.

#### To create the XML file

- Create a new XML file that describes the fields to export from and import into the entity. Name the file *EntityNameFlow.xml* and save it to the `modules/configuration/config/resources/exportimport` subdirectory of your PolicyCenter installation directory.  
Use the `CPLocationFlow.xml` and `CPBuildingFlow.xml` files as a model for creating additional `EntityInfo` XML files.

- Add the following root element to the file:

```
<EntityInfo
 EntityTypeName="entity.EntityName"
 ParentEntityTypeName="entity.parentEntityName"
 ParentEntityColumnPath="EntityName.ID">
```

Attribute	Required?	Description
EntityTypeName	Required	The name of the entity preceded by: <code>entity.</code>
ParentEntityTypeName	Optional	If the parent entity is also to be imported and exported, the name of the parent entity preceded by: <code>entity.</code>
ParentEntityColumnPath	Optional	The path that identifies the unique ID column of the parent entity, if a parent entity is specified.

3. Within the EntityInfo element, add a collection of ColumnInfo child elements. Each ColumnInfo element specifies one column to be exported to and imported from the spreadsheet. The order of the ColumnInfo elements determines the order of the columns in the exported spreadsheet.

```
<ColumnInfo
 ColumnType="pathTypeName"
 ExcludeFromTemplate="true|false"
 FlagsAction="true|false"
 FlagsEntityId="true|false"
 Header="displayKeyPath"
 id="columnName"
 Locked="true|false"
 Path="beanPathToColumnValue"
 RequiredForImport="true|false">
```

Attribute	Required?	Description
ColumnType	Optional	The full path type name of the column's data value. If the column type cannot be handled by one of the predefined data column resolvers, you must define a new data column resolver for each such type. Then adjust the ColumnDataResolverFactory class to select the new type when appropriate. Default: String.
ExcludeFromTemplate	Optional	Whether to exclude the column when exporting a template. Default: false.
FlagsAction	Optional	Whether the column specifies the action for the spreadsheet row. Only one column can have this attribute set to true, and this is the <b>Action</b> column, typically exported as the first column in the spreadsheet. All other attributes except Header are ignored. Default: false.
FlagsEntityId	Optional	Whether the column uniquely identifies the entity. Default: false.
Header	Required	The display key path that contains the string that is output as the column header. Using a display key enables the column headers to be localized. Specify the Header value for the Action column as Export.Action. PolicyCenter writes the column display keys under Export → Entity, though a display key for that name may already exist at some other location. You can quickly look at all exported column data under the Export node.
Id	Optional	The name identifier of the column upon which other columns are dependent. Used only when another ColumnInfo specifies a Required element with this ColumnInfo as its target. For more information, see "Dependent Elements" on page 508.
Locked	Optional	Whether the column value cannot be modified. Locked columns are set to read-only and shaded gray in the exported spreadsheet. Default: False.
Path	Optional	Required except when FlagsAction is true. The bean path that accesses the column value of the entity.
RequiredForImport	Optional	Whether the column value must be specified when the spreadsheet is imported. Default: false.

## Parent Element

If the entity you are defining is a child of another entity that is to be exported and imported, add a Parent element within the EntityInfo element. Within the Parent element, add a map of the parent's ColumnInfo elements. For example:

```
<EntityInfo ...>
.
.
.
<Parent>
 <ColumnInfo
 Path="CPLocation.PublicID"
 Header="Export.CPBuilding.LocationID"
 Locked="true"
```

```

 ColumnType="java.lang.String" />
<ColumnInfo
 Path="CPLocation.Location.LocationName"
 Header="Export.CPBuilding.LocationName"
 ColumnType="java.lang.String" />
...
</Parents>
.
.
.
</EntityInfo>
```

## Dependent Elements

A `ColumnInfo` element can also contain an optional map of `Dependent` child elements, each of which specifies another column on which the containing `ColumnInfo` is dependent. Each `Dependent` element takes a single attribute, `purpose`, that is a key to the id of a `ColumnInfo` on which the containing `ColumnInfo` is dependent. For example:

```

<!-- Column that can have dependent columns because it has an id attribute -->
<ColumnInfo
 Path="CPLocation.Location.State"
 Header="Export.CPBuilding.State"
 ColumnType="typekey.State"
 id="stateColumn"
 RequiredForImport="true" />
.
.
.
<!-- Column that depends on another column -->
<ColumnInfo
 Path="ClassCode"
 Header="Export.CPBuilding.ClassCode"
 ColumnType="entity.CPClassCode"
 RequiredForImport="true" >
 <Dependent purpose="State">stateColumn</Dependent>
</ColumnInfo>
```

## Step 2: Define Column Data Resolvers

Each type (class) other than `String` to be exported and imported must have its own `ColumnDataResolver` class. This class specifies how to convert a value to a string on export and convert the string back into the correct value type on import. PolicyCenter defines the following `ColumnDataResolvers`:

- `NullColumnDataResolver` – Imports nothing. Typically used for the `Action` column and for any informational column that is processed during import or export operations.
- `SimpleColumnDataResolver` – Exports and imports strings and numbers. It is so named because processing strings and numbers is simple compared to other types.
- `TypeKeyColumnDataResolver` – Exports a resolved type key as a string and imports a string as a type key.
- `CoverageColumnDataResolver` – Exports the display value of a coverage term as a string. Imports the localized coverage term value using the dotted path to resolve the coverage pattern and coverage term pattern.
- `TerritoryCodeColumnDataResolver` – Exports a territory code value as a string and imports a string as a territory code value.
- `TaxLocationColumnDataResolver` – Exports a tax location value as a string and imports a string as a tax location value.
- `ClassCodeColumnDataResolver` – Exports a class code value as a string and imports a string as a class code value.
- `AbstractColumnDataResolver` – An abstract class that uses iterative reflection to resolve the dotted bean path to the value that needs to be exported and imported. Typically you extend this class when defining new column data resolvers.
- `ColumnDataResolver` – An interface that is responsible for importing and exporting cell data values. Implemented by `AbstractColumnDataResolver`.

If you need to export and import a type (class) that is not handled by one of the provided column data resolvers, you must implement a new one. In Studio navigate to **configuration → gsrc** and define a new class in `my_domain_name.exportimport.resolver` where `my_domain_name` is the package for your company's files. Give the class a name appropriate to the column type it is designed to resolve.

Typically it is sufficient to leverage the `AbstractColumnDataResolver` and implement the method `#calculateCellValue`. This method constructs and returns a new object of a type that matches the `ColumnInfo` type.

The following code shows how to implement a new `TaxLocationColumnDataResolver`. If the `ColumnInfo` needs more context to resolve the value, it is possible to define dependent data for the `ColumnInfo`, as illustrated in bold in the example. In this case, the `TaxLocationColumnDataResolver` requires state information to calculate the jurisdiction and find the `TaxLocation`. This dependency must be completed by using the `Dependent` element and `id` attribute in the `EntityInfo` XML file, as explained in “[Dependent Elements](#)” on page 508.

```
uses
.
.
.
class TaxLocationColumnDataResolver extends AbstractColumnDataResolver<TaxLocation> {
 construct(aColumnInfo : ColumnInfo) {
 super(aColumnInfo)
 }

 override function calculateCellValue(bean : KeyableBean, cellData : CellData,
 metaData : ImportMetaData, dependentData : Map<String, Object>) : TaxLocation {
 var period = (bean as EffDated).BranchUntyped as PolicyPeriod
 if (cellData.Data.NotBlank) {
 var state = dependentData["State"] as State
 if (state == null) {
 createIssue(bean, cellData, metaData,
 displaykey.Import.Validation.Errors.MissingDependentDataState)
 }

 var searchCriteria = new TaxLocationSearchCriteria()
 searchCriteria.Code = cellData.Data
 searchCriteria.State =
 StateJurisdictionMappingUtil.getJurisdictionMappingForState(state)
 searchCriteria.EffectiveOnDate = period.EditEffectiveDate
 var foundTaxLocations = Lookups.getTaxLocation(searchCriteria)
 if (foundTaxLocations.Empty) {
 createIssue(bean, cellData, metaData,
 displaykey.Import.Validation.Errors.NoTaxLocationsFound(cellData.Data,
 state.DisplayName))
 }
 return foundTaxLocations.first()
 }
 return null
 }
}
```

### Update the `ColumnInfo` class

For each new column data resolver, edit the `ColumnInfo` class to select the new type when appropriate. To open this class, use Studio to navigate to **configuration → gsrc** and open the class `gw.exportimport.resolver.ColumnDataResolverFactory`.

Map each new column data resolver class to an appropriate type in the `TYPE_RESOLVER_MAP`.

## Step 3: Create an Import Strategy for Each New Data Column Resolver

The import strategy defines how (and whether) to create and delete entities during an import operation. To create an import strategy:

1. In Studio, navigate to **configuration → gsrc** and define a new class in `gw.exportimport.entityimport`. Name the class to reflect the entity you are configuring. For example, if you are configuring the `PersonalVehicle` entity for import and export, name the new class `PersonalVehicleImportStrategy`.

2. Implement the `deleteEntity` and `createEntity` methods as needed, using return values that are appropriate to the entity.
3. Specify a true or false return value for the `AllowCreate` and `AllowDelete` properties. If the entity you are configuring has complex deletion requirements, consider prohibiting delete during import.  
Examine the `CPLocationImportStrategy.gs` and `CPBuildingImportStrategy.gs` as models for creating new import strategies.
  - `CPLocationImportStrategy` illustrates how to allow creation of new entities and prevent deletion. Deleting locations is prohibited. PolicyCenter does not allow deleting a location that is in use, and there is no easy way to determine if a location is in use during an import operation.
  - `CPBuildingImportStrategy.gs` illustrates a somewhat more complex scenario of creating buildings within a location as well as the scenario of deleting a building.

## Step 4: Modify the ImportStrategyRegistry Class

The final implementation step is to register your new import strategies. To register an import strategy:

1. In Studio, navigate to `configuration` → `gsrc` and edit the class  
`gw.exportimport.entityimport.ImportStrategyRegistry`.
2. In the `TYPE_STRATEGY_MAP` variable declaration, map the entity you are configuring to your new import strategy. For example:

```
class ImportStrategyRegistry {
 private construct() {
 }

 /**
 * Map of {@link EntityImportStrategy}'s by Entity <code>Type</code>.
 */
 static final var TYPE_STRATEGY_MAP : Map<Type, EntityImportStrategy> = {
 CPBuilding -> new CPBuildingImportStrategy(),
 CPLocation -> new CPLocationImportStrategy()
 PersonalVehicle -> new PersonalVehicleImportStrategy()
 }
}
```



## chapter 37

# Configuring Earned Premium

Earned premium is the portion of premium that applies to the expired part of the policy period. In other words, earned premium is the amount of premium that has been earned as of the current date. This topic describes how to configure the earned premium which appears on the **Earned Premium** field on the policy **Summary** screen.

For reporting policies, the calculation includes the earned-but-unreported (EBUR) amount prior to final audit. For package policies, PolicyCenter displays earned premium for each line of business.

This topic includes:

- “How PolicyCenter Calculates Earned Premium” on page 511
- “Methods that Calculate Earned Premium” on page 512
- “PCF Files that Display Earned Premium” on page 513

**See also**

- “Policy Tools Menu” on page 309 in the *Application Guide*
- “Policy Earned Premium Web Services” on page 113 in the *Integration Guide*

## How PolicyCenter Calculates Earned Premium

The earned premium calculation examines all financial transactions on the policy. The calculation checks a series of conditions, most of which are based on the calculation, or *as-of*, date. For reporting policies where EBUR may be included, it also uses the date of the last premium report period. The calculation evaluates a transaction as follows:

1. If the *as-of* date is prior to either the posted date or the written date of the transaction, then the earned amount for the transaction is 0.
2. Else if the amount type on the related cost is not premium, for example taxes or surcharges, then the earned amount for the transaction is 0.
3. Else if the transaction is to be accrued (*ToBeAccrued* is *true*), then:
  - a. If the *as-of* date is after the expiration date of the transaction, then the whole transaction amount counts toward earned premium.
  - b. Else if the *as-of* date is prior to the effective date of the transaction, then the earned amount for the transaction is 0.

- c. Else, the only other possibility is that the as-of date is between the transaction's effective and expiration dates. The earned amount is calculated as a prorated portion based on the as-of date.
4. Else, the only transactions left are transactions that are not to be accrued (`ToBeAccrued` is `false`). These transactions are of two types:
- a. If the transaction is earned on the effective date of the policy, The whole transaction amount counts toward earned premium. Transactions for flat-rated costs are an example.
  - b. Else if the transaction is for a cost that is subject to reporting, the transaction must meet both of the following criteria.
    - 1) The calculation includes EBUR.
    - 2) And the as-of date is after the date of the last premium reporting period.For these transactions, the earned amount is the prorated portion between the last premium report date and the as-of date.

#### See also

- “Prorated or Flat Costs” on page 425 in the *Application Guide*

## Methods that Calculate Earned Premium

An enhancement on the `Transaction` entity, `TransactionEPEnhancement.gsx`, defines methods to calculate earned premium. You can modify these methods.

### Earned Premium

The `earned` method calculates the earned premium for the transaction as of the current date. The method signature is:

```
earned(lastReportedDate: Date, includeEBUR: boolean): BigDecimal
```

This method calls the `earnedAsOf` method with the `asof` parameter set to the current date. The parameters are described in that method.

### Earned Premium As of Date

The `earnedAsOf` method calculates the earned premium as of a specific date passed in as a parameter.

```
EarnedAsOf(asof, lastReportedDate, includeEBUR)
```

- `asof` – The as-of date for determining the earned premium.
- `lastReportedDate` – The latest date for which a premium report was processed (the end date of the report). This parameter is only applies to reporting policies.
- `includeEBUR` – Set to `true` to calculate EBUR on the current transaction. This parameter only applies to reporting policies.

The calculations for earned premium include or exclude EBUR based on the `includeEBUR` parameter. In the base configuration, PolicyCenter EBUR is configured as follows:

- The PCF that calculates earned premium automatically sets `includeEBUR` to `true` for reporting policies where the final audit is not complete. For more information, see “PCF Files that Display Earned Premium” on page 513.
- For all other policies `includeEBUR` is set to `false`.

Through configuration, you can change whether PolicyCenter includes EBUR in the calculation.

## PCF Files that Display Earned Premium

The `Policy_Summary_EarnedPremiumDV.pcf` displays earned premium. For reporting policies, this screen shows whether or not the earned premium includes EBUR. The **Earned As Of** date is a link that enables you to change the as-of date for the calculation.



# Configuring the Team Tab

This topic describes how to configure the **Team** tab.

This topic includes:

- “Configuring the Team Screens Batch Process for Team Statistics” on page 515
- “Setting the Window Size for Team Statistics” on page 516
- “How PolicyCenter Calculates Reporting Categories” on page 517
- “Setting the Maximum Number of Rows on the Team Screens” on page 518

## Configuring the Team Screens Batch Process for Team Statistics

The **Team Screens** batch process generates the **Summary** statistics that the **Team** tab screens display. In the default configuration, the **Team Screens** batch process runs once an hour. If the summary statistics are out-of-date and you need to see more recent information, you can run the batch process manually. See “Running Batch Processes” on page 103 in the *System Administration Guide*.

**Note:** Through configuration, you can create additional categories for statistics. However, the **Team Screens** batch process will not generate statistics for those categories. The **Team Screens** batch process only calculates statistics for activities, submissions, renewals, and other work orders.

When configuring the **Team Screens** batch process, consider the following:

- How frequently the batch process runs. See “Scheduling the Team Screens Batch Process” on page 516 for details.
- The time window used to calculate user statistics. See “Setting the Window Size for Team Statistics” on page 516.
- The logic for calculating the summary statistics. See “How PolicyCenter Calculates Reporting Categories” on page 517

## Scheduling the Team Screens Batch Process

Set the frequency that the Team Screens batch process runs in `scheduler-config.xml` file. This batch process gets and compiles the data for the statistics on the Team tab. The following lines in the `scheduler-config.xml` file set the frequency of the Team Screens batch process:

```
<!-- Hourly user statistics generation at three minutes past the hour -->
<ProcessSchedule process="TeamScreens">
 <CronSchedule minutes="3"/>
</ProcessSchedule>
```

The time and database workload required for running the Team Screens batch process may be significant and may adversely impact system performance if run too frequently.

### See also

- “Scheduling Work Queues and Batch Processes” on page 107 in the *System Administration Guide*

## Setting the Window Size for Team Statistics

The window sets the boundary for submissions, activities, or other reporting categories that the Team tab displays. The window applies to all columns except the Open column. The Calculated on field is the time when the batch process ran and is the end of the window. The start of the window is calculated backwards from that time. In the following illustration, the window for activities and other reporting categories are circled in red.

In the default configuration, the reporting categories for activities and work orders are for one week.

You can set the window for reporting categories to the following sizes:

Window size	Description
This week	<p>The window starts at the beginning of the current business week until the most current run of the Team Screens batch process.</p> <p>The start of current business week is the end of the last business week. In the base configuration, the end of a business week is configured as Friday at 5:00 p.m. Therefore, the start of the current business week is 5:00 pm of the Friday prior to the current day. If that day is a holiday, then PolicyCenter goes back to the last known business day.</p>
This month	<p>The start of the current month until the most current run of the Team Screens batch process.</p> <p>The start of the current month is the first day of the current month at 12:00 a.m.</p>
N days	<p>The window is the last N days of activity, including the current day.</p> <p>The start of a day is 24 hours prior to the most current run of the Team Screens batch process. Therefore, for a one day window, if the batch process runs at 7/11/08 10:54 a.m., the start of the window is 7/10/08 10:54 a.m.</p>

You can configure the statistics window for all reporting categories except activities.

In Studio, you configure the window size in config.xml by modifying the following parameters:

Parameter name	See also
OtherWorkOrdersStatisticsWindowSize	"OtherWorkOrdersStatisticsWindowSize" on page 49
RenewalsStatisticsWindowSize	"RenewalsStatisticsWindowSize" on page 49
SubmissionsStatisticsWindowSize	"SubmissionsStatisticsWindowSize" on page 49

## How PolicyCenter Calculates Reporting Categories

The following tables describes how PolicyCenter calculates which activities, jobs, or work orders appear in each reporting category depending upon the window. PolicyCenter calculates the status when it runs the Team Screens batch process. The Team tab displays the time that the batch process ran in the Calculated on field.

**Note:** You cannot change how PolicyCenter calculates the status of reporting categories.

### Activities with a One Day Window Example

The Calculation column in the following table describes how PolicyCenter calculates the status for each activity in the Activities reporting category.

The Example column describes which activities will be included. The example assumes that the statistics were last calculated at 7/11/08 10:54 a.m. and that the activities window is one day.

Status	Calculation	Example
Open	Activity with open status	Includes any open activity. The window does not apply.
Overdue	Activity with open status and target date on or before the start of the window.	Includes an open activity due on or before 7/10/08 10:54 a.m.
Completed	Activity with completed status and close date on or after to the window start date.	Includes an activity completed between 7/10/08 10:54 a.m. and calculating the statistics.

### Submissions with a This Week Window Example

The default window size for activities and work orders is this week.

The Calculation column in the following table describes how PolicyCenter calculates the status for each submission in the Submission by User reporting category.

The Example column describes which submissions will be included. The example assumes that the statistics were last calculated at 07/11/08 10:54 a.m. and that the submissions window is this week. The 07/11/08 date is a Friday. In the base configuration, the end of the business week is Friday at 5:00 p.m. Therefore, the business week started on Friday, 07/04/08 at 5:00 p.m.

Status	Calculation	Example
Open	The job close date field is not set.	Includes any open submission. The window does not apply.
New	The job close date is not set, and the job create time is on or after the window start date.	Includes a submission if it was created between 7/4/08 5:00 p.m. and calculating the statistics.
Bound	Any policy period associated with the job has a non-null model number. The job close date is on or after the window start date.	Includes a submission bound between 7/4/08 5:00 p.m. and calculating the statistics.

## Renewals with a This Month Window Example

The Calculation column in the following table describes how PolicyCenter calculates the status for each renewal in the **Renewals by User** reporting category.

The Example column describes which renewals will be included. The example assumes that the statistics were last calculated at 7/11/08 10:54 a.m. and that the renewal window is **this month**. The month started on 7/1/08 at 12:00 a.m.

Status	Calculation	Example
Open	The job close date field is not set.	Includes any open renewal for this user. The window does not apply.
New	The job close date is not set, and the job create time is on or after the window start date.	Includes a renewal created between 7/01/08 12:00 a.m. and calculating the statistics.
Renewed	Any policy period associated with the job has a non-null model number. The job close date is on or after the window start date.	Includes a renewal issued between 7/01/08 12:00 a.m. and calculating the statistics.
Non-Renewed	Any policy period on the job is locked and the status field is non-renewed. Also, the job close date is on or after the window start date.	Includes a renewal non-renewed between 7/01/08 12:00 a.m. and calculating the statistics.
Not-Taken	Any policy period on the job is locked and the status field is not taken. Also, the job close date is on or after the window start date.	Includes a renewal not taken between 7/01/08 12:00 a.m. and calculating the statistics.

## Other Work Orders

The **Other work orders** category includes cancellations, reinstatements, policy changes, rewrites, rewrite new accounts, and audit jobs.

The Calculation column in the following table describes how PolicyCenter calculates the status for each work order in the **Other work orders** reporting category.

Status	Calculation
Open	The job close date field is not set.
New	The job close date is not set, and the job create time is on or after the window start date.
Approved	Any policy period associated with the job has a non-null model number. The job close date is on or after the window start date.

## Setting the Maximum Number of Rows on the Team Screens

You can use the `TeamScreenTabVisibilityRowCountCutoff` configuration parameter to set the maximum number of rows that the **Activities**, **Submissions**, **Renewals**, and **Other Work Orders** screens shows on the **Team** tab. Increasing this parameter potentially increases the amount of time that PolicyCenter takes to render the **Team** screens. If the results exceed the value of this parameter, then PolicyCenter displays a message on that screen and does not display the results.

If you select an individual user, PolicyCenter always displays the filtered results even if the number of results exceeds the cutoff parameter.

Using submissions as an example, view the **Summary** screen. If the total number of **Open**, **New**, or **Bound** submissions is greater than value of this parameter, then PolicyCenter does not display any submissions on the **Submissions** screen. PolicyCenter displays a message on the **Submissions** screen.

**See also**

- “TeamScreenTabVisibilityRowCountCutoff” on page 50



# Configuring Rating Management

Guidewire Rating Management has been designed to be extended through configuration if required. However, major configuration changes may not be required.

---

**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.

---

This topic includes:

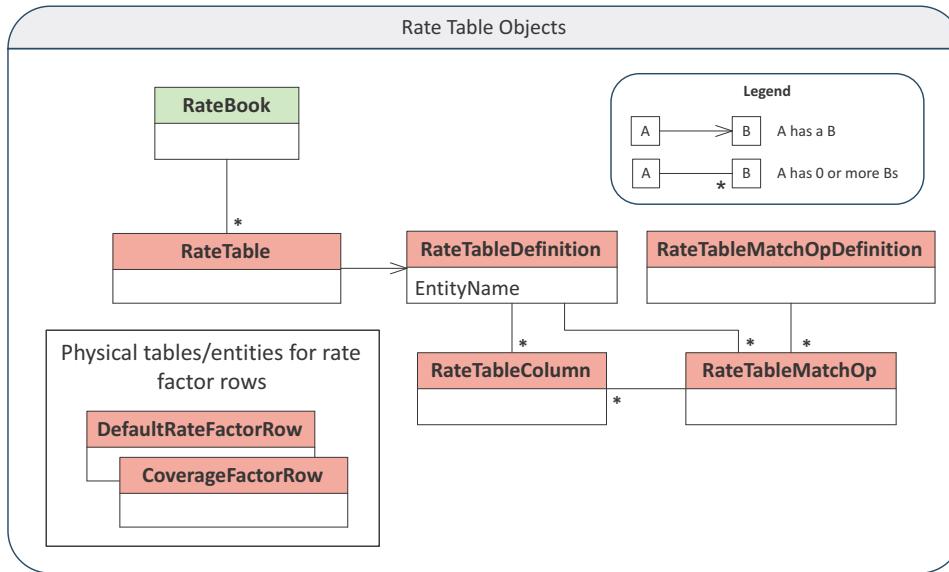
- “Rating Management Data Model” on page 521
- “Improving Rate Table Performance” on page 523
- “Parameters and Properties for Rating Management” on page 524
- “Third Party Libraries for Rating Management” on page 526
- “User Authority and Permissions for Rating Management” on page 526
- “Custom Physical Tables for Rating Management” on page 527
- “Configuring Value Providers” on page 528
- “Configuring Matching Rule Operations” on page 530
- “Configuring Rate Routines” on page 533
- “Configuring New Parameters in Parameter Sets” on page 540
- “Configuring Rating Worksheets” on page 543
- “Configuring Impact Testing” on page 546
- “Rating Management Plugins and Interfaces” on page 548

## Rating Management Data Model

This topic describes the data model for Guidewire Rating Management.

## Rate Table Data Model

The following illustration shows some of the object relationships for rate tables.



The **RateTable** entity represents a rate table. The rate table has a pointer to a **RateTableDefinition**.

The **RateTableDefinition** entity represents a rate table definition. The **EntityName** property contains the name of the physical table or entity of a rate factor row. The rate factor row entity describes the data types that you can assign to parameters and factors.

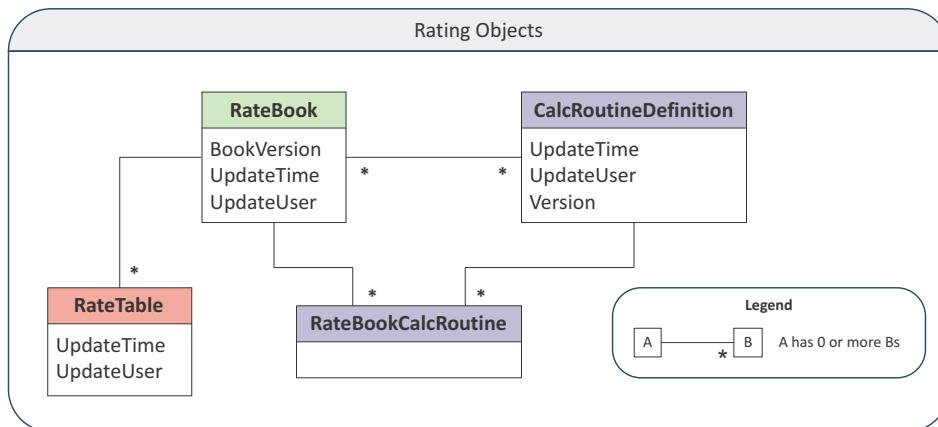
In the default configuration, the rate factor row entities are **DefaultRateFactorRow** and **CoverageRateFactor**.

### See also

- “Physical Tables and Entities for Rate Table Definitions” on page 543 in the *Application Guide*

## Rate Routine Data Model

The following illustration shows some of the object relationships for rate routines.

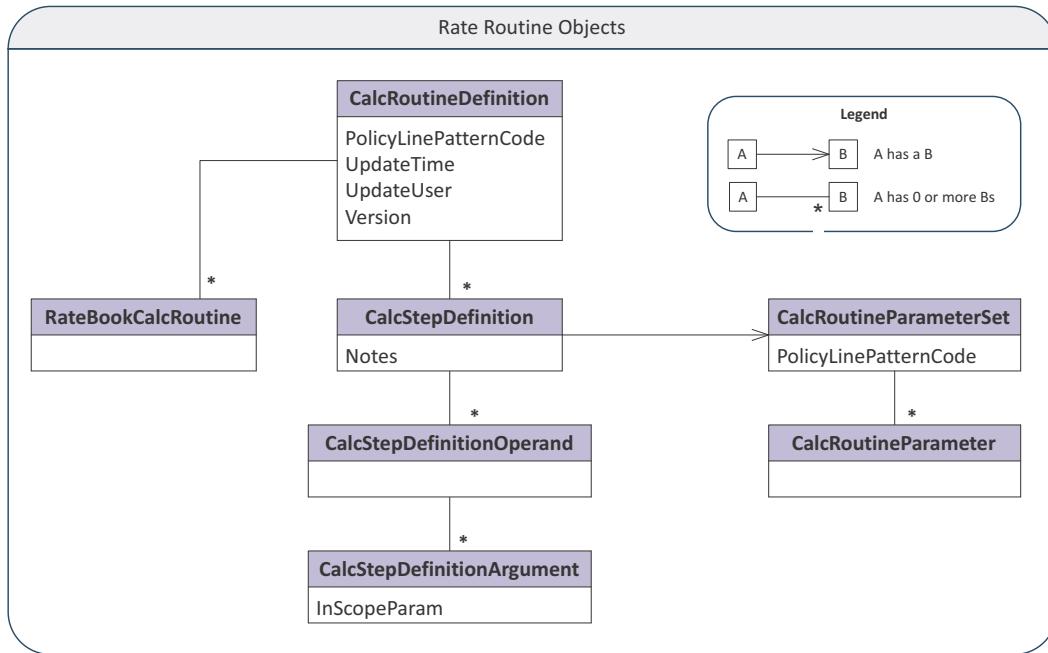


The **RateBook** object represents a rate book and contains rate tables and rating routines.

The **RateTable** object represents a rate table.

The `CalcRoutineDefinition` object represents a rate routine definition. In PolicyCenter, you define rate routines on the **Administration** → **Rate Routines** screen.

The `RateBookCalcRoutine` object represents a rate routine included in a rate book. In PolicyCenter, rate routines included in a rate book appear on the **Included Rate Routines** tab on the **Rate Book Details** screen.



A rate routine is composed of multiple steps, represented by the `CalcStepDefinition` object.

## Improving Rate Table Performance

For each rate table, you can choose whether to load the table into memory or access it from the database. By default, PolicyCenter loads rate tables into memory. Loading the rate table in memory can provide quick access. Accessing the table on disk is slower, but loading a large table into memory may slow system performance even more.

To decide whether or not to load the table into memory, consider how large the table will be and how often it will be referenced. Consider database lookup if:

- The table has multiple thousands of rows and is referenced infrequently. For example, referenced only on a small percentage of quotes.
- The table is very large (multiple tens of thousands of rows) and is only referenced once or twice per quote.

For anything smaller, or for a table referenced numerous times to quote a policy, use memory lookup, if possible. Database lookup can result in significant slowdown.

Rating management consults the lookup setting when promoting the rate book status from **Draft** to **Stage**. Therefore, you can only change the lookup setting when the rate book is in **Draft** status.

A rate table can be included in multiple rate books but can have only one setting for lookup. You can only change the lookup setting in the first rate book that included the rate table, if it is in **Draft** status.

Performance depends on the rate table content, not just the row count. For a table that is small enough to fit in memory, in-memory will usually outperform a call out to the database. For database-queried tables, if the query tends to hit the correct row every time, database performance is better than if the query frequently goes through the relaxing process. A query hitting on the first try usually corresponds to a table which is mostly filled out, with very few null values. On the other hand, you can expect relaxation to occur often if the table has many null values.

#### See also

- “Rate Table Lookup in Memory or Database” on page 544 in the *Application Guide*
- “Including Rate Tables in a Rate Book” on page 566 in the *Application Guide*

## Covering Index to Improve Performance

Doing repeated database queries against a generic physical table (such as `DefaultRateFactorRow`) can cause performance issues.

If a rate table is large enough to warrant running the queries in the database, configure the physical table to have a custom row entity with a *covering index*. The covering index on the row entity includes the foreign key to the `RateTable`, the `Retired` property, and all parameters and factors.

For example, you have a custom row entity, `CustomFactor`. This row contains properties for parameters named `OptCode` and `Jurisdiction`. The index for this row begins with the foreign key to rate table and `Retired`, followed by the parameters and factors:

```
<index
 name="factorlookup"
 desc="Speeds lookup queries against this table.">
 <indexcol
 name="RateTable"
 keyposition="1"/>
 <indexcol
 name="Retired"
 keyposition="2"/>
 <indexcol
 name="OptCode"
 keyposition="5"/>
 <indexcol
 name="Jurisdiction"
 keyposition="6"/>
 <indexcol
 name="Factor"
 keyposition="7"/>
</index>
</entity>
```

#### See also

- “Physical Tables and Entities for Rate Table Definitions” on page 543 in the *Application Guide*
- “Custom Physical Tables” on page 544 in the *Application Guide*

## Parameters and Properties for Rating Management

This topic describes parameters and properties related to Rating Management.

- “Minimum Rating Level Parameter” on page 524
- “Rate Table Normalization Configuration Parameters” on page 525
- “Logging Properties for Rating Management” on page 525

### Minimum Rating Level Parameter

Rating Management has a configuration parameter to set the minimum rating level.

In production, only rate books in **Active** status are used for policy rating. However, since active rate books can never be updated, it is useful to be able to test with rate books not yet in **Active** status. Then you can make changes in the current version instead of creating another version of the rate book for the next cycle of testing.

Rating Management has a **RatingLevel** parameter to support rating with rate books not yet in **Active** status in a development or test environment. This parameter controls the minimum rate book status that the rating query considers as a candidate. The default level is **Active**, which is the recommended level for production environments. A lower level can be set for development or test environments.

The rating level is set in a parameter on the **IRatingPlugin** plugin in `gw.plugin.policyperiod`. The parameter can have one of the following **RateBookStatus** values (in decreasing order):

- **Active**
- **Approved**
- **Stage**
- **Draft**

The **RatingLevel** is passed to the query as part of the filter. The query considers all books that are of equal or higher status than the minimum rating level.

#### See also

- “Guidewire Rating Management and PCRatingPlugin” on page 355 in the *Integration Guide*

## Rate Table Normalization Configuration Parameters

In a rate table that contains a range parameter and any other parameter, the table may have ranges that overlap. For tables that fall within specified configuration parameters, PolicyCenter normalizes the rate table to speed up rate factor lookups.

In `config.xml`, two parameters control whether or not PolicyCenter normalizes a rate table by default. These parameters are:

- **RateTableManagementNormalizationRowThreshold**

Default: 1000

If the number of rows in the normalized table would exceed this threshold, the user is given the option to store a non-normalized version of the table.

- **RateTableManagementNormalizationRowLimit**

Default: 10000

PolicyCenter determines whether to normalize a rate table if the table contains this many rows or more.

If the number of rows in the normalized table would exceed this limit, the table is automatically marked as non-normalizable. PolicyCenter stores the non-normalized version of the table.

#### See also

- “Rate Table Normalization” on page 544 in the *Application Guide*

## Logging Properties for Rating Management

Rating Management defines the following logging categories:

Category	Description
<code>Application.Rating</code>	Logging for messages related to Guidewire Rating Management.
<code>Application.Rating.RateTableManagement</code>	Logging for messages related to rate table management.
<code>Application.Rating.Rateflow</code>	Logging for messages related to creating and executing rate routines.

**See also**

- “Configuring Logging” on page 23 in the *System Administration Guide*

## Third Party Libraries for Rating Management

Rating Management has the following third-party libraries.

### POI Library

The POI library supports reading and writing rate tables to and from Microsoft Excel workbooks. POI is an Apache licensed library. This library provides Java APIs for manipulating various file formats based upon the Office Open XML standards (OOXML) and Microsoft's OLE 2 Compound Document format (OLE2).

## User Authority and Permissions for Rating Management

The default configuration includes a starter set of permissions and roles related to Guidewire Rating Management. Review and update these as required for your specific implementation. See “Security: Roles, Permissions, and the Community Model” on page 647 in the *Application Guide* for details on roles and permissions.

The default configuration has the following permissions and roles.

### Rating Management Permissions

The permissions related to Rating Management are:

- `ratebookview`
- `ratebookedit`
- `ratebookapprove`
- `editrateasofdate`
- `usereditlang`
- `rateimpacttesting`

### Rating Management Roles

The roles related to Rating Management are:

- `rating_analyst`
- `rating_supervisor`

## Assignment of Permission to Roles

Permissions are assigned to roles in the default configuration as follows:

Permission	Code	Assigned to roles
View rate books and tables	ratebookview	superuser rating_analyst rating_supervisor underwriter underwriter_supervisor
Edit rate books and tables	ratebookedit	superuser rating_analyst rating_supervisor
Approve rate books and tables	ratebookapprove	superuser rating_supervisor
Edit Rate as of Date	editrateasofdate	superuser underwriter underwriter_supervisor
Edit user language	usereditlang	superuser rating_analyst rating_supervisor underwriter underwriter_supervisor
Rate policies for rate impact testing	rateimpacttesting	superuser rating_analyst rating_supervisor

## Custom Physical Tables for Rating Management

The default configuration includes two physical tables that can be used as-is and as models for other custom tables that may be required for a particular implementation. These tables are:

- A default, generic physical table that can be used to store many of the rate tables required for most lines of business. This table is the `DefaultRateFactorRow` entity.
- A sample custom physical table for simple coverage-based rate tables. This table is the `CoverageRateFactor` entity.

If neither table is appropriate, you can configure a new custom physical table for one or a group of similar rate table definitions.

### See also

- “Physical Tables and Entities for Rate Table Definitions” on page 543 in the *Application Guide*

## Configuring a New Custom Physical Table

To configure a new custom physical table, you must create an `.eti` file for a system entity. You can use the `CoverageRateFactor` entity as a model for your new custom entity.

The following requirements must be satisfied when creating a custom entity for rate table factors:

- Must have a foreign key to `RateTable` entity.

- The name of the foreign key must be `RateTable`.
- Must define at least two columns to accommodate at least one parameter and at least one factor.
- Must have at least one column of either `String`, `Integer`, `Decimal`, `Boolean`, or `Date` type. These are the types supported for parameters and factors.

The custom entity must be created in the `modules\configuration\config\extensions` directory.

The custom entity can be referenced in the **Physical Table** field in the **Basics** tab of the **Rate Table Definition Editor**.

## Configuring Value Providers

The default configuration includes the following types of value providers:

- “Typelist Value Provider” on page 528
- “PolicyCenter Product Model Value Providers” on page 528
- “Reference Factor Value Provider” on page 529

If none of these is appropriate, a new custom value provider can be configured.

All classes related to value providers are located in the `gw.rating.rtm.valueprovider` package. A value provider may use an optional list of string arguments. Value providers are instantiated through the `RateTableCellValueProviderFactory` factory class.

This topic describes the value providers in the default configuration, and how to configure a new value provider.

### See also

- “Value Provider” on page 542 in the *Application Guide* for user-level information

### Typelist Value Provider

The typelist value provider is:

- Implemented in `TypelistValueProvider` class.
- Tied to a typelist defined in the system.
- Does not support any optional attributes.
- A list of values is sorted according to typelist priorities.

### PolicyCenter Product Model Value Providers

The PolicyCenter product model value providers are:

- “Coverage Value Provider” on page 528
- “Coverage Term Value Provider” on page 529
- “Coverage Term Option Value Provider” on page 529
- “Termless Coverage Value Provider” on page 529

**Note:** Availability of coverages, terms and options is not checked by any of these value providers.

### Coverage Value Provider

The coverage value provider, `CoverageValueProvider`:

- Extends the abstract `AbstractProductModelValueProvider` class.
- Provides a list of policy line coverages for a particular policy line.
- Returns:

- A list of coverage codes for values
- A composite label consisting of coverage category name and coverage name based on a string template defined in the `Web.Rating.CoverageValueProvider.DisplayLabel` display key
- The list is sorted by coverage category priority first and then by coverage priority.
- Does not support any optional attributes.

### Coverage Term Value Provider

The coverage term value provider, `CovTermValueProvider`:

- Extends abstract `AbstractProductModelValueProvider` class.
- Provides a list of coverage terms for specific policy line coverage.
- Returns a list of coverage term codes sorted by priority.
- Supports one optional argument: coverage code.
- Use for rate table columns satisfying one of the following conditions:
  - A rate table column depends on another column that contains coverage code.
  - An argument is specified for this value provider.

### Coverage Term Option Value Provider

The coverage term option value provider, `CovTermOptionValueProvider`:

- Extends the abstract `AbstractProductModelValueProvider` class indirectly through the `AbstractCovTermOptionValueProvider` class.
- Provides a list of coverage term options for a specific coverage term of a specific coverage.
- Returns a list of coverage term option codes sorted by priority.
- Supports two optional parameters: coverage code and coverage term code
- Works with either none or both parameters.
- Use for rate table columns satisfying one of the following conditions:
  - A rate table column depends on another rate table column that contains coverage term code.
  - Both arguments are specified for this value provider.

### Termless Coverage Value Provider

The termless coverage value provider, `TermlessCoverageValueProvider`:

- Is similar to `CoverageValueProvider` but returns only those coverages that have no terms.

## Reference Factor Value Provider

The reference factor value provider:

- Is implemented in the `ReferenceFactorValueProvider` class.
- Retrieves all distinct values used in a column of a rate table (source rate table).
- Must have two arguments configured:
  - Rate table code – Code of a source rate table.
  - Column name – Name of the column in the source rate table
- Returns a list of distinct values from a specific column (second argument) of the source rate table (first argument).
- Does not return any specific labels, just the values themselves.
- Implements internal cache (a hash map): `(argument 1 + argument 2) -> ({set of values})`.

- Initialized when a value provider is constructed.
- Populated for each combination of argument values on each call to `ReferenceFactorValueProvider#getValues()` method for those combinations of arguments that are not found in the cache.
- The lookup of the source rate table is done in the same rate book as the table specifying this value provider.

## Creating a New Value Provider

If none of the default value providers is appropriate, you can configure a new custom value provider.

You create new value providers by extending the `RateTableCellValueProvider` abstract class. Create new value providers in the `gw.rating.rtm.valueprovider` package. The user interface has the following restriction: the package is hard-coded and does not display in the rate table definition user interface.

Performance considerations have to be taken into account when creating new value providers. Values may be retrieved from value providers multiple times when the **Rate Table Editor** screen loads. Therefore, caching responses that the value provider gets from certain sources may improve performance of the user interface. For an example of caching in value providers, see any of the product model value providers.

For the new value provider to appear in the list of custom value providers in the **Rate Table Definition Editor**, you must add a new typecode to the `ValueProvider` typelist. The **Code** of the added typecode must be the same as the name of the value provider Gosu class.

## Configuring Matching Rule Operations

In a rate table, you define a matching rule for each parameter. You set the matching rule to one of the match operations defined in the application.

If none of the default match operations is appropriate, you can configure a new custom match operation.

### See also

- “Matching Rules” on page 571 in the *Application Guide*

## Configuring a New Match Operation

To configure a new match operation:

1. In Product Designer, navigate to the `rtm_matchop_defs.xml` system table.
2. In `RateTableMatchOpDefinition`, add an entry to the system table. You must specify the following:
  - **OpCode** – A code that uniquely identifies this match operation.
  - **OpName** – Appears in PolicyCenter as a match operation choice when the user clicks **Add** in the **Parameters** tab of the **Rate Table Definition Editor** screen.
  - **NumberOfParameters** – The number of parameters.
  - **ImplClass** – The fully qualified name of the new match operation factory class. For example, you can define the class as `gw.rating.matchop.OpCodeMatchOpFactory`.
3. Create the following new classes:
  - A factory class that extends abstract `gw.rating.matchop.MatchOperationFactory`. See “Extending the Match Operation Factory Class” on page 531.
  - A class that extends abstract `gw.rating.matchop.StatelessMatchOperator`. See “Match Operation Implementation” on page 531.
4. Optionally, create a custom validator for the match operation.

- A class that extends `gw.rating.validation.MatchOpValidator`. See “Match Operation Validator” on page 532.

## Extending the Match Operation Factory Class

The match operation factory class, `MatchOperationFactory`, is an abstract base class that provides a static method to get match operation factory by name: `getFactoryByName(String)`. You must create a class that extends `gw.rating.rtm.matchop.MatchOperationFactory`. For an example, see `gw.rating.rtm.matchop.LessThanOrEqualMatchOpFactory.gs`.

Your new match operation factory implementation must implement at least the following two methods.

### Method That Creates a Stateless Match Operator

This method signature is:

```
function createStatelessMatchOperator(entity.RateTableMatchOp): <S extends StatelessMatchOperator>
```

This method must create and return a new instance of the match operation implementation class as described in “Match Operation Implementation” on page 531.

### Method That Creates a Validator

This method signature is:

```
function createValidator():MatchOpValidator
```

This method must create and return a new instance of the match operation validator class as described in “Match Operation Validator” on page 532.

## Match Operation Implementation

You must create a class that extends abstract `gw.rating.matchop.StatelessMatchOperator`. For an example, see `gw.rating.rtm.matchop.StatelessLessThanOrEqualMatch`.

The match operation implementation constructor accepts the following argument:

- An instance of `RateTableMatchOp` entity. This instance is part of the rate table definition.

Your new match operation implementation can provide implementations of methods from the base class, including the following methods.

### Filter That Evaluates Row Under Consideration

This filter method signature is:

```
function filter(List<OrderedPersistenceAdapter>, Comparable, boolean):List<OrderedPersistenceAdapter>
```

This in-memory filter evaluates all rows still under consideration, and returns only those rows which satisfy the conditions for the match operation. This filter is only called when the match operation is active. A corresponding relax method is not needed, because the list of results to be filtered are pre-partitioned according to whether the row would match a filter or a relax. The input results must be fully ordered and stay that way upon output. For more information about relaxing, see also “Matching a Factor in the Rate Table” on page 545 in the *Application Guide*.

A stateless match operation doing in-memory filtering expects its input to be in ascending order with respect to the field or fields being matched by the match operation. For example, a range match operation matches the fields `int1` (minimum) and `int2` (maximum). This operation assumes that the rows are ordered according to increasing `int1` value, with a tie being broken by comparing the `int2` values.

The output of the stateless match operation must be a list that is properly formatted for the next match operation. Therefore, the output must be in ascending order according to the field or fields matched by that next match operation.

To meet this requirement, PolicyCenter sorts the rate table when it is first loaded, using order by/then by sorting. The rate table is sorted by the first match operation's columns. Then each region with equal values for the first match operation is sorted for the second match operation's columns, and so on. A match operation controls how its rows are ordered by overriding the method:

```
compareRowValues(a : Comparable, b : Comparable) : int
```

If a match operation does not naturally produce correctly-ordered output from this form of input, it needs to reorder the rows as part of its filtering process. For most match operations, an easy way to do this is to override the property `needsSubrangeMerge` so that it returns `true`. The `StatelessLessThanOrEqualMatch` provides an example of this.

### Filter That Adds a Constraint to the Database Query

This filter method signature is:

```
function filter(Query<KeyableBean>, Comparable<Object>)
```

Adds a constraint to the database query and returns the adjusted query.

Entity instances that are passed into the above functions represent rate factor entities such as `RateFactorRow`, `CoverageRateFactor`, or any other custom entity.

### Method Which Computes a Score

This method signature is:

```
function getScore(OrderedPersistenceAdapter, Comparable) : Comparable
```

Used to compute a score for the input argument, `Comparable`, in relation to rate table row value, `OrderedPersistenceAdapter`. This score is used during rate table matching to determine the best match.

## Match Operation Validator

The match operation validator validates the parameter inputs to the match operation. To implement a validator, you must create a class that extends abstract `MatchOpValidator` in the `gw.rating.rtm.validation` package. For an example validator see `RangeMatchOpMaxInclValidator` class in the `gw.rating.rtm.validation` package.

**Note:** If the match operation does not require a custom validator, you can use the `NoOpValidator` when creating the `MatchOperationFactory` for the new match operation. For an example, see `ExactMatchOpFactory` in the `gw.rating.rtm.matchop` package.

A new match operation validator must implement the following methods.

### Validate Method

This method signature is:

```
function validate(List<Comparable>) : boolean
```

Returns `true` if all values are valid from the point of view of this match operation. For example, an exact match accepts any value, so it always returns `true`. See step 4 in “Configuring a New Match Operation” on page 530.

A range match operation accepts a list of two values where either value can be null or the first value is less than the second value. Otherwise, return `false`.

### Validate Pair Method

This method signature is:

```
function validatePair(List<Comparable>, List<Comparable>) : boolean
```

Validates two sets of values against each other and returns `true` if both sets are valid with regards to each other.

This method is mainly used to identify overlaps and other value conflicts. For example, for the range match operation validator, this function checks if two numeric (or date) ranges have no overlap.

## Configuring Rate Routines

Rate routines implement rating algorithms that calculate the properties on the cost for coverages, taxes, and other costs on a policy. This topic describes how to configure rate routines:

- “Configuring the Rating Engine to Execute the Rate Routine” on page 533
- “Adding a New Rate Routine Function” on page 534
- “Configuring Rounding Operators” on page 535
- “Configuring Prefixes that Identify Types in Rate Routine Steps” on page 535
- “Configuring Variant Identifiers for a Rate Routine” on page 536
- “Configuring the Rate Routine Plugin” on page 538
- “Using Gosu to Get the Rate Factor from the Rate Table” on page 538

### Configuring the Rating Engine to Execute the Rate Routine

You can add a rate routine in the PolicyCenter Rate Routines screen. However, that rate routine is not part of rating a policy until you configure the rating engine to execute the rate routine.

In Rating Management, the built-in rating plugin implementation is `PCRatingPlugin`. This class implements rating only for the personal auto and commercial property lines of business. The sample data in the default configuration contains rate routines for these lines of business. The rate routines provided in the default configuration are already configured in the rating engine.

This topic describes how to extend the `PCRatingPlugin` class to execute new rate routines that you add.

#### See also

- “Guidewire Rating Management and `PCRatingPlugin`” on page 355 in the *Integration Guide* for instructions on how to enable the `PCRatingPlugin` plugin and extend it to other lines of business.
- “Adding a New Rate Routine” on page 578 in the *Application Guide*

### Extending the Rating Engine to Execute New Rate Routines

If you add new rate routines in PolicyCenter, you must extend the rating engine for the line of business specified in the rate routine. The `PCRatingPlugin` class calls the rating engine for each line of business. In the default configuration, the rating engine classes that use rating management are `PARatingEngine` and `CPRatingEngine`.

**IMPORTANT** When adding code that executes a rate routine, be aware that simply executing a rate routine consumes both time and resources, whether or not the rate routine performs any actions. Therefore, only execute the rate routine when necessary. For example if a rate routine applies to a specific coverage on a policy, then make sure your code executes the rate routine only when the policy contains that coverage. The `executeCalcRoutine` method executes the rate routine.

#### To extend the rating engine to execute new rate routines

In the rating engine, add code that calls the `RateBook.executeCalcRoutine` method. For example, to execute a new rate routine in personal auto, modify the `PARatingEngine` class in the `gw.lab.pa.rating` package.

The method signature for `executeCalcRoutine` method is:

```
function executeCalcRoutine(code : String, costData : CostData<Cost, PolicyLine>,
 worksheetContainer : WorksheetEntryContainer,
 paramSet : Map<CalcRoutineParamName, Object>)
```

The arguments are:

- `code` – The rate routine code that uniquely identifies the rate routine, as a `String`.

- `costData` – The cost data object which contains properties you can use to calculate a rate, as a `CostData`.
- `worksheetContainer` – Holds information for rating worksheets. Rating worksheets describe the series of calculations to generate a rate, as a `WorksheetEntryContainer`. For more information, see “Rating Worksheets” on page 558 in the *Application Guide*.
- `paramSet` – A map representing the parameters in the parameter set for the rate routine, as a `java.util.Map`.

If the rate routines does not calculate properties on the cost, set the `costData` parameter to `null` and the `worksheetContainer` to a `NoCostWorksheetContainer`. For example, the following code in `PARatingEngine.gs` executes the `pa_assign_driver_style1_rr` rate routine:

```
RateBook.executeCalcRoutine("pa_assign_driver_style1_rr", null,
 new NoCostWorksheetContainer(), assignDriverParameterMap)
```

The `PARatingEngine` class is in the `gw.lob.pa.rating` package. The `NoCostWorksheetContainer` stores data about what the rate routine executed. However, the default implementation of rating worksheets does not save this information to the database nor display it on the **Rating Worksheet** screen.

#### See also

- “Rate Routines That Do Not Calculate Properties on the Cost” on page 550 in the *Application Guide*

### Example of How the Rating Engine Executes the Rate Routine

The sample data for the personal auto line of business contains a `PA Vehicle Coverage Premium Algorithm` rate routine. This rate routine provides a cost for vehicle level coverages on the policy. The code for this rate routine is `pa_veh_cov_premium_rr`.

The `PCRatingPlugin` class calls the `PARatingEngine` class to rate a personal auto policy. The `PARatingEngine` class is in the `gw.lob.pa.rating` package.

In the `PARatingEngine` class, the `rateVehicleCoverage` method rates vehicle-specific coverages such as a coverage with the `PALiabilityCov` code. If the policy has this coverage, the code executes the `pa_veh_cov_premium_rr` rate routine.

### Adding a New Rate Routine Function

A rate routine function is a predefined function that you can include in a rate routine step. Rate routine functions are defined in Gosu classes. You can define your own functions by adding a Gosu method to these classes. The function can return a value of any type. Functions with a return value appear in the **Operand** column of a rate routine step. Void functions appear in the **Instruction** column. Void functions do not return a value. An example of a void function is `logAmount`.

The `SharedRatingFunctions` class in the `gw.rating.flow.util` package contains rating functions for use across product lines.

For line-specific functions, create a class that extends the class `RatingFunctionsSource`. By extending this class, PolicyCenter calls your rate routine function class. Functions in this class appear on the policy lines specified in the `availableForLine` method. This method takes a policy line code `String` and returns `true` if the class supports that policy line. This class must implement the `availableForLine` method.

For example, the `PARatingFunctions` class in the `gw.lob.pa.rating` package implements rate routine functions only for the personal auto line.

A rate routine function cannot be a parameterized method. If you define a rate routine function as a parameterized method, it does not appear in the list of functions on the **Select a Function** screen. PolicyCenter sends a warning to the log file. For more information about parameterized methods, see “Parameterized Methods” on page 245 in the *Gosu Reference Guide*.

#### See also

- “Logging Rate Routine Functions in a Rating Worksheet” on page 546

## Configuring Rounding Operators

In the default configuration, you can use the following rounding operators in a rate routine:

- **R** – Round half up. Round towards the nearest number according to scale. If both numbers are equidistant, round away from 0.
- **RD** – Round down. Round down to the nearest number according to scale. Always round towards 0.
- **RU** – Round up. Round up to the nearest number according to scale. Always round away from 0.
- **RE** – Round half-even. Round towards the nearest number according to scale, unless both numbers are equidistant. If equidistant, round towards the even number.

Through configuration, you can make the following optional rounding operators available in rate routines:

- **RC** – Round ceiling. Round up to positive infinity.
- **RF** – Round floor. Round down to negative infinity.
- **HD** – Round half-down. Round towards nearest number according to scale, unless both numbers are equidistant. If equidistant, round towards 0.
- **NR** – Unnecessary. The requested operation is expected to produce an exact result. Throws an exception if rounding is necessary.

### See also

- “Rounding Operators” on page 582 in the *Application Guide*

### Making Optional Rounding Operators Available in Rate Routines

This topic describes how to configure PolicyCenter to make the optional rounding operators available for use in rate routine steps. In the default configuration, PolicyCenter the optional rounding operators, **RC**, **RF**, **HD**, and **NR**, are not available.

In Studio, the rounding operators are in the **rounding** category. Optional rounding operators are in the **optrounding** category.

This topic describes how to make the **RF** optional rounding operator defined by the **floor** typecode available.

1. In Studio, open the `CalcStepOperatorType.tti` typelist. The **File** tab lists the typecodes in the typelist.
2. Click the **Categories** tab.
3. Expand the **optrounding** category in the **Categories** hierarchy.  
This category contains the optional rounding operators. The **floor** operator is in this category.
4. Return to the **File** tab.
5. In the **Element** hierarchy, expand the **floor** typecode to reveal the category contained within.  
For the **floor** operator, the category value is **optrounding**.
6. In the **Element** hierarchy, click **category** in the **floor** typecode.
7. Change Value of the code to **rounding**.
8. Click the **Categories** tab to verify that **floor** now appears in the **rounding** category.

## Configuring Prefixes that Identify Types in Rate Routine Steps

For some instruction and operand types, you can add or change the prefix that appears in the rate routine step. For example, in the default configuration, **table:** precedes the name of the rate table in a step. When changing the prefix, be aware that the prefix takes up space in the rate routine step.

In the default configuration, Studio includes display keys for prefixes on properties on the cost, function, rate table, and variable types.

### To change the type prefix for properties on the cost, functions, rate tables, and variables

1. In Studio, navigate to configuration → config → Localizations → *Lang* → *display.properties*.
2. In Display Keys, find the display key you wish to modify.

For each type, the following table contains the name of the display key, the default prefix, alternate prefix, and the unicode value of the alternate prefix. The alternate prefix is provided as a suggestion only. You can define the prefix to a string of your choice. You can enter the unicode character directly into the Studio editor.

Instruction and operand type	Display key	Default prefix	Alternate prefix	Unicode value of alternate
Properties on the cost	Web.Rating.Flow.CalcRoutine.CostDataStoreLabel	None		
Function	Web.Rating.Flow.CalcRoutine.RateFunctionLabel	None	j	222B
Rate table	Web.Rating.Flow.RateTableLabel	table:		229E
Variable	Web.Rating.Flow.CalcRoutine.RateVariableLabel	None	v:	

If you add a variable prefix, the prefix appears for user defined variables and multiple factor variables.

3. Select the locale from the drop-down list.
4. Modify the prefix in the **Locale** text box.
5. Select **File** → **Synchronize Product Model** to update PolicyCenter.
6. In PolicyCenter, navigate to a rate routine to view the new prefix.

#### See also

- “Instruction and Operand Types” on page 583 in the *Application Guide*
- “Using the Display Keys Editor” on page 137

## Configuring Variant Identifiers for a Rate Routine

A rate routine can have variant rate routines. The rate routine and its variant rate routines have the same code. In the default configuration, the jurisdiction variant is an example of a variant identifier. Through configuration, you can extend or change the definition of variants to have other identifiers. For example, you can configure a variant identifier to other features of the policy such as the underwriting company. Through configuration, you can remove the jurisdiction variant. For more information, see “Rate Routine Variant Identifiers” on page 551 in the *Application Guide*.

This topic describes how to configure a variant identifier for a rate routine.

- “Add the variant identifier to the data model” on page 536
- “Add the rate routine variant identifier to the rate routine plugin” on page 537
- “Add the rate routine variant identifier to PCF files” on page 537

#### Add the variant identifier to the data model

In Studio, add a column with the name of the variant identifier to *CalcRoutineDefinition.eti* and include the name in the index. For detailed instructions, see “Extending a Base Configuration Entity” on page 215.

For example, add a column for the variant identifier named `NewVariantIdentifier` defined as follows:

Name	Value
name	<code>NewVariantIdentifier</code>
type	<code>shorttext</code>
nullok	<code>true</code>
desc	A new variant identifier
localization	<code>a_table_name</code>

And add a `NewVariantIdentifier` index to `CalcRoutineDefinition` defined as follows:

Name	Value
name	<code>NewVariantIdentifier</code>
keyposition	4

Change the `Retired` index `keyposition` from 4 to 5. The `NewVariantIdentifier` index is inserted before `Retired`.

#### Add the rate routine variant identifier to the rate routine plugin

The getter for the `BranchingFields` property returns an ordered list of zero or more variant identifiers. In the implementation of `IRateRoutinePlugin`, add the name of the variant identifier to the `fields` variable of the getter for the `BranchingFields` property.

The built-in implementation of the rate routine plugin is `RateRoutinePlugin.gs` in the `gw.plugin.rateflow` package.

For example, in your implementation of the rate routine plugin, add `NewVariantIdentifier` to the `fields` variable as follows:

```
override property get BranchingFields() : IEntityPropertyInfo[] {
 var fields : ArrayList<String> = {"Jurisdiction", "NewVariantIdentifier"}
 ...
}
```

#### Add the rate routine variant identifier to PCF files

In Studio, add the rate routine variant identifier to the following PCF files.

1. Add New Variant Identifier to the Rate Routine Details and New Rate Routine screens.

- a. Add a `Web.Rating.RateRoutines.NewVariantIdentifier` display key and set the value to New Variant. You can add the display key by adding it to the `configuration → config → Localizations → lang → display.properties` file.
- b. In `RateRoutineDV.pcf`, add a basic input after the `Jurisdiction` input. For example, from the `Toolbox`, go to `Basic Inputs` and insert an `Input` with the following properties:

Property	Value	Description
<code>editable</code>	<code>canEditIdentifyingFields</code> or <code>copyVersionType == BRANCH</code>	Whether the input is editable.
<code>id</code>	<code>NewVariantIdentifier</code>	A unique identifier for this widget.
<code>label</code>	<code>displaykey.Web.Rating.RateRoutines.NewVariantIdentifier</code>	The input label.
<code>value</code>	<code>rateroutine.NewVariantIdentifier</code>	Set value to the <code>NewVariantIdentifier</code> property on the <code>CalcRoutineDefinition</code> entity instance.

2. On the **Rate Routine Details** screen, change the name of the **Create Jurisdiction Variant** button to something more descriptive such as **Create Variant Rate Routines**. This button will now create variants where you can set **Jurisdiction** and **New Variant**.
3. Add a **New Variant** column in the **Search Results** on the **Rate Routines** screen.
  - a. In **RateRoutinePanelSet.pcf**, add a basic cell just after the **Jurisdiction** cell. For example, go to **Basic Cells** and insert a **Cell** with the following properties:

Property	Value	Description
id	RateroutineNewVariantIdentifier	A unique identifier for this cell.
label	displaykey.Web.Rating.RateRoutines.NewVariantIdentifier	The cell label.
value	rateroutine.NewVariantIdentifier	Set value to the <b>NewVariantIdentifier</b> property on the <b>CalcRoutineDefinition</b> entity instance.

4. On the **Rate Book Details** screen, add **New Variant** to the **Included Rate Routines** tab.
  - a. In **RateBookDetailsScreen.pcf**, add a **TextCell** just after **Jurisdiction** with the following properties:

Property	Value	Description
id	NewVariantIdentifier	A unique identifier for this cell.
label	displaykey.Web.Rating.RateRoutines.NewVariantIdentifier	The cell label.
value	rateroutine.NewVariantIdentifier	The cell value.

5. Add **New Variant** to other rate routine screens where the **Jurisdiction** variant appears.

#### See also

- “Rate Routines” on page 550 in the *Application Guide*
- “Get Branching Fields” on page 552

## Configuring the Rate Routine Plugin

When PolicyCenter executes a rate routine, PolicyCenter calls the rate routine plugin defined by the **IRateRoutinePlugin** interface. The rate routine plugin:

- Enables rating worksheets in the plugin.
- Sets up cost data wrappers that determine which properties on the cost appear when you select an instruction or operand in a rate routine step.

For more information, see “Rate Routine Plugin” on page 549.

## Using Gosu to Get the Rate Factor from the Rate Table

You can get the rate factor from a rate table in a rate routine. In certain circumstances however, getting the rate factor may involve more complex logic than rate routines support. This topic describes how to get the rate factor from the rate table using Gosu code. You might add this code in the rating engine or in a rate routine function.

This topic provides an example of how to accomplish this using workers’ compensation as an example.

#### Create a class for getting the rate factor

Create a class to contain methods that get the rate factors from all rate tables in this line of business. For workers’ compensation, you can define a **WCRateFactorSearch.gs** class in the **gw.lab.wc.rating** package.

### Define a method to get the rate factor

Follow these steps to define a `getFactor` method that gets the rate factor from a rate table. This method is general purpose and is not specific to a particular line of business. If you are adding Rating Management to multiple lines of business, you could choose to put this method in its own class.

1. In the public method for the class, add private variables for storing the policy period and rate book status.

```
class WCRateFactorSearch {

 private var _period : PolicyPeriod as readonly Period
 private var _minRatingLevel : RateBookStatus as MinimumRatingLevel

 construct() {

 }
```

2. Insert uses statements after the package statement at the top of the file. The method that you define in the following steps uses these classes.

```
package gw.lob.wc.rating

uses java.math.BigDecimal
uses gw.rating.rtm.query.RateQueryParam
uses java.lang.Comparable
uses java.util.Date
uses gw.rating.rtm.query.RateBookQueryFilter
uses gw.job.RenewalProcess
uses gw.rating.rtm.query.RatingQueryFacade

class WCRateFactorSearch {
```

3. After the class constructor, add the `getFactor` method that gets the rate factor.

```
construct() {

}

private function getFactor<Q extends Comparable>(jurisdiction : Jurisdiction,
 inputParams : List<RateQueryParam<Q>>,
 tableCode : String,
 ratingDate : Date) : BigDecimal
{
 /* Constructs a new rateBookQueryFilter object and sets properties such as Jurisdiction */
 /* and UWCompany. */
 var filter = new RateBookQueryFilter(ratingDate, Period.RateAsOfDate,
 Period.WorkersCompLine.PatternCode)
 {
 :Jurisdiction = jurisdiction,
 :UWCompany = Period.UWCompany,
 :Offering = Period.Offering.Code,
 :MinimumRatingLevel = MinimumRatingLevel,
 :RenewalJob = Period.JobProcess typeis RenewalProcess
 }
 var result = new RatingQueryFacade().getFactor<Q, BigDecimal>(filter, tableCode, inputParams)
 return !result.Empty ? result.Factor : BigDecimal.ONE
}
```

The input parameters to the `getFactor` method are:

- `jurisdiction` – The jurisdiction of the policy location.
- `inputParams` – Specify values for each parameter in the rate table.
- `tableCode` – The code of the rate table from which to obtain the rate factor. This is the `TableCode` property on the `RateTableDefinition` object. In the user interface, `tableCode` is the `Code` on the `Basics` tab of the `Rate Table Definition` screen.
- `ratingDate` – The date for which to obtain rates. The date is compared against the `RateBook.EffectiveDate` or `RateBook.RenewalEffectiveDate` properties depending upon the job type. In the user interface, `ratingDate` is the `Policy Effective` or `Coverage Reference Date` or `Renewal Effective Date` in `Policy Criteria` on the `Rate Book Details` screen.

This method return the rate factor as a `BigDecimal`. If no rate factor is found, the method return the value 1.

### Define methods that get the rate factor

In this series of steps, you define a method that gets the rate factor. The rate factor is used to calculate rates for ratable objects such as coverables. Repeat these steps to define a method for each ratable object. This example gets the rate factor for a workers' compensation class code.

1. If you must get the jurisdiction from a policy location, add a uses statement at the top of the file after the package name.

The method that gets the jurisdiction for a particular ratable object uses methods in this class.

```
uses gw.api.util.JurisdictionMappingUtil
```

2. In the public method for the class, add a private variable for storing the code of the rate table. The code is passed to the getFactor method in the tableCode parameter.

```
class WCRateFactorSearch {
```

```
 private var WC7_CLASS_CODE_TABLE = "WorkersComp_ClassCode_Rate"
```

The private variable is not strictly necessary. Instead of using this private variable, you could pass the tableCode parameter directly in the call to the getFactor method.

3. Add a method that gets the rate factor for a particular ratable object. This method calls the getFactor method. Name the method appropriately for the type of ratable object.

In this example, the method is getClassCodeRate. This method gets the rate for a covered employee's class code from the rate table.

```
function getClassCodeRate(covEmp : WCCoveredEmployee, ratingDate : Date) : BigDecimal {
 var jurisdiction
 JurisdictionMappingUtil.getJurisdictionMappingForPolicyLocation(covEmp.LocationWM)
 var params = {
 new RateQueryParam<String>("CLASSCODE", covEmp.ClassCode.Code)
 }
 return getFactor(jurisdiction, params, WC7_CLASS_CODE_TABLE, ratingDate)
}
```

The input parameters to this method vary depending upon the ratable object. The input parameters to the getClassCodeRate method are:

- covEmp – The WCCoveredEmployee object from which to get the class code.
- ratingDate – The date for which to get the rate.

This method returns the rate factor as a BigDecimal.

## Configuring New Parameters in Parameter Sets

This topic describes how to create a new parameter and make the parameter available for inclusion in a parameter set. The new parameter is available in PolicyCenter, and you can insert the new parameter in rate routine steps. For example, in **Administration** → **Rating** → **Parameter Sets**, select a parameter set. In the **Parameters** panel, click **Edit**. To add a new parameter, click **Add**. In the **Name** column, drop-down list contains the list of parameters. This topic describes how to add a parameter to the list.

In PolicyCenter, parameter sets are associated with rate routines and rate table definitions. In addition, you must add the parameter to the rating engine code so that the parameter can be used in rating.

To configure a parameter:

- **In the PolicyCenter user interface, create the parameter for inclusion in a parameter set.** Define the parameter and specify a default type, such as entity.PersonalVehicle for a personal vehicle entity. Then you can include the parameter in a parameter set. In the parameter set, you can override the parameter's default type. Each rate routine that uses that parameter set can access that parameter in the rate routine steps.

For instructions, see “Creating a New Parameter for Inclusion in Parameter Sets” on page 541.

- **In Studio, modify the rating engine so that it rates the object or entity instance that the parameter specifies.** For each rate routine that includes the parameter set, add a mapping from the parameter to an object type. The object type could be the primary vehicle on a personal auto policy. When the rating engine executes the rate routine, the parameter is used to reference the actual run-time instance or value. The run-time instance could be the primary vehicle on Solomon's personal auto policy.  
For instructions, see "Adding a New Parameter to the Rating Engine" on page 542.
- **If the parameter uses a wrapper, create the wrapper code.**  
For instructions, see "Configuring New Wrappers for Parameter Sets" on page 543.

The parameter definitions must meet the following criteria:

- **Parameter types must be compatible.** In the parameter set, the parameter's type must be compatible with the Gosu object that the rating engine passes to the method that executes the rate routine. If the user overrides the parameter's default type in the parameter set, the override takes precedence. The rating engine must pass an object that is the same type or a subtype of the parameter type. PolicyCenter throws an exception if the rating engine passes in an object that is not compatible with the parameter associated with the executing rate routine.
- **In the rating engine, the parameter map must be a superset of the rate routine parameters.** When the rating engine executes a rate routine, one of the arguments is a map of the rate routine's parameters. In the rate routine parameter map, each parameter is mapped to an object or entity instance, such as a vehicle on Solomon's policy. The rate routine parameter map must contain all of the names in the parameter set defined in PolicyCenter for the rate routine being executed. The parameter set may contain the names of additional parameters. The rating engine throws an exception if it executes a rate routine and the parameter map is missing any parameters in the parameter set.

#### See also

- "Adding a New Parameter to the Rating Engine" on page 542
- "Adding, Changing, or Removing Parameters from a Parameter Set" on page 592 in the *Application Guide*
- "Parameter Sets" on page 552 in the *Application Guide*
- "Filtering Parameter Set Fields of the Same Type When Entering Parameters in a Rate Routine" on page 585 in the *Application Guide*

## Creating a New Parameter for Inclusion in Parameter Sets

These instructions describe how to create a new parameter for inclusion in parameter sets and accessible in rate routines in the PolicyCenter user interface. To create a new parameter, you must complete the following instructions:

- "To add the parameter to the parameter name typelist for rate routines" on page 541
- "To enter the default type for the parameter" on page 542
- "To include the parameter in a parameter set in PolicyCenter" on page 542

#### To add the parameter to the parameter name typelist for rate routines

Create a new typekey for the parameter. In a parameter set, the parameter **Name** appears in the **Name** column on the **Parameter Sets** → **Parameters** tab in PolicyCenter.

1. In Studio, navigate to **configuration** → **config** → **Metadata** → **Typelist** and open `CalcRoutineParamName.tti`.
2. Click typelist at the top of the **Element** hierarchy.

3. Click the drop-down list next to  and choose `typecode`. This adds a typecode for the new parameter to this typelist. For example, you can add a `PATest` parameter defined as follows:

code	name	desc
patest	PATest	Personal Auto Test

#### To enter the default type for the parameter

Enter the default type which appears in the **Type** column on the **Rating → Parameter Sets → Parameters** tab in PolicyCenter. If you do not specify a default type, the **Type** column is blank.

1. In Studio, open `RatingParameterSetScreenHelper.gs` in the `gw.pcf.rating.flow` package.

2. In the `mapFromCodeToType` static variable, enter a mapping for the code of the new parameter.

Enter the code in capital letters prefixed by `CalcRoutineParamName.TC_`. Enter the default type as the fully qualified object type. For example:

```
CalcRoutineParamName.TC_PATEST -> "entity.PersonalAutoLine"
```

#### To include the parameter in a parameter set in PolicyCenter

In the PolicyCenter user interface, include the parameter in one or more parameter sets. When you add the parameter, you can override the default type. Keep in mind that the object the rating engine passes must be compatible with the type of object that rate routine expects. For instructions and more information, see “Adding, Changing, or Removing Parameters from a Parameter Set” on page 592 in the *Application Guide*.

## Adding a New Parameter to the Rating Engine

These instructions describe how to add a new parameter to the rating engine. When the rating engine runs a rate routine, the rating engine maps each parameter to an object or entity instance. The map is of type `java.util.Map`. The rating engine uses this map to pass run-time objects and entity instances to the method that executes the rate routine. This object or entity instance must be compatible with the type specified in the parameter set associated with the rate routine being executed. A compatible type is the same type or a subtype. If the object or entity instance is not compatible, then PolicyCenter throws an exception.

**Note:** For instructions on how to add a new parameter for use in PolicyCenter, see “Configuring New Parameters in Parameter Sets” on page 540.

In these steps, you include the new parameter in Gosu code that executes each rate routine associated with this parameter set.

The rating engine code executes each rate routine by calling the `RateBook.executeCalcRoutine` method. One of the parameters to this method is a map of parameters that the rate routine uses. This map must be a superset of the parameter set in PolicyCenter. By definition, a superset contains at least all of a set.

#### To include the parameter in the rating engine code

For each rate routine associated with the new parameter’s parameter set, you must add the new parameter to the map. The parameter map is in the rating engine class in Studio.

1. In Studio, open the rating engine class for the line of business that uses this parameter. For example, open `PARatingEngine.gs` if this parameter is for the personal auto line. The `PARatingEngine` class is in the `gw.lob.pa.rating` package.
2. Modify the `executeCalcRoutine` method call for each rate routine associated with a parameter set that includes this parameter.

The following steps assume that you added a parameter to the `PIP NJ Parameter Set` parameter set. The rate routine with the `pa_pip_nj_basic_rr` code uses this parameter set.

The code that executes this rate routine is:

```
RateBook.executeCalcRoutine("pa_pip_nj_basic_rr",
 :costData = data,
 :worksheetContainer = data,
 rateRoutineParameterMap)
```

The `rateRoutineParameterMap` argument is a map defined as:

```
var rateRoutineParameterMap : Map<CalcRoutineParamName, Object> =
{TC_POLICYLINE ->_line,
 TC_VEHICLE ->vehicle,
 TC_PAPIPNJ ->cov}
```

In the map, the `Object` must be compatible with the type for the parameter set being used by the executing rate routine. You specify the type in the `Type` column on the `Parameter Sets → Parameters` tab in the PolicyCenter user interface.

3. In the `rateRoutineParameterMap` variable, add a map entry for your new parameter.

4. Start or restart PolicyCenter to view your changes.

## Configuring New Wrappers for Parameter Sets

You may have parameters sets that are essentially the same, but each line of business uses a different coverage, for example. Using *wrappers*, you can combine these parameters sets into one parameter set that uses a wrapper to select the coverage by line of business.

The wrapper calls Gosu code that selects a coverage based upon characteristic of the policy. You must first create the wrapper before you use it in a parameter set.

In Studio, create a new Gosu class that implements `gw.rating.flow.CoverageWrapper`. For an example, see `CPCoverageWrapper.gs` in the `gw.lob.cp.rating` package.

### See also

- “Combine Similar Parameter Sets with Wrappers” on page 553 in the *Application Guide*

### Example with Wrappers

In the base configuration, the **CP Coverage Parameter Set With Wrapped Coverages** parameter set (`CPCoverageWrapperParamSet`) uses a wrapper. The **Coverage** parameter uses the wrapper `gw.lob.cp.rating.CPCoverageWrapper` to select either the **Business Personal Property Coverage** (`CPBPPCov`) or the **Building Coverage** (`CPBldgCov`). In Product Designer, you can view these coverages in the **Commercial Property Line**.

The **CP Coverage Premium Algorithm** rate routine (`cp_cov_premium_rr`) uses the **CP Coverage Parameter Set With Wrapped Coverages** parameter set. In this rate routine, the **Basis** is assigned the value of `Coverage.Limit`. The wrapper calculates the value of the limit based upon whether the coverage is **Business Personal Property Coverage** or **Building Coverage**.

## Configuring Rating Worksheets

In the default configuration, rating worksheets are configured for the personal auto and commercial property lines of business. To configure rating worksheets for other lines of business, you must do the following:

- Configure rating worksheets in the rate routine plugin** – For more information, see “Enabling Rating Worksheets in the Rate Routine Plugin” on page 549.
- Configure the user interface** – Implement the `getWorksheetRootNode` method in the `policy-line-methods` class for the line of business. This method controls how the worksheets are organized in the user interface. See `PAPolicyLineMethods.gs` class as an example. In this example, the code that traverses the cost version list is similar to the code that displays the costs in the **Policy Premium** tab of the **Quote** screen.

The user must have the **View rating worksheet** permission to view a rating worksheet. The code for this permission is `ratingworksheetview`.

**See also**

- “Rating Worksheets” on page 558 in the *Application Guide*

## Configuring Extract and Purge Rating Worksheets

Every time a user generates a rating worksheet, a copy of that rating worksheet gets stored in the PolicyCenter database. To improve PolicyCenter performance, PolicyCenter provides processes for extracting rating worksheet data and removing rating worksheet containers from the database. This is not an end user feature and is only accessible through **Server Tools**. The Extract Worksheets batch process extracts the rating worksheet data to files and marks worksheet containers for purging. The Purge Rating Worksheets batch process removes the worksheet containers from the database. You must enable this batch process.

This topic describes how to configure these two batch processes and the associated plugins.

**See also**

- “Extracting and Purging Rating Worksheets” on page 559 in the *Application Guide*

### Extract Rating Worksheets Batch Process

**Code – extractworksheets**

The Extract Rating Worksheets batch process extracts rating worksheet data from worksheet container (`WorksheetContainer`) objects to files in a specified directory on the batch server. The batch process also marks `WorksheetContainer` objects for purging.

In the base configuration, this batch process is not scheduled to run on a regular basis. You can run it manually or schedule it.

In the base configuration, the batch process extracts worksheet data to files to an `extracted_worksheets` directory off the file system root. Extract Rating Worksheets processes `WorksheetContainer` objects attached to a `Policy` on which all jobs are closed. Closed jobs have a non-null `Job.CloseDate`. The batch process does not process `WorksheetContainer` objects that have already been extracted.

In Studio, you can change the extract destination directory by modifying `WorksheetExtractPlugin.gwp`. Modify the value of the `WorksheetExtractDestination` parameter.

In Studio, you can modify the behavior of the batch process by modifying the worksheet extract plugin. For more information, see “Worksheet Extract Plugin” on page 545.

### Purge Rating Worksheet Batch Process

**Code – purgeworksheets**

The Purge Rating Worksheets batch process removes worksheet container (`WorksheetContainer`) objects that meet both of the following criteria:

- Are marked for purging
- That are on policies whose jobs have been closed more than a specified number of days (90 days in the base configuration).

In the base configuration, the Extract Rating Worksheets batch process marks the worksheets container objects for purging.

In the base configuration, this batch process is disabled. After you enable it, you can run it manually or schedule it.

A `WorksheetContainer` is marked for purging if its `CanPurge` flag is `true`.

The `RatingWorksheetContainerAgeForPurging` parameter in `config.xml` specifies the minimum number of days after a job is closed before the `WorksheetContainer` associated with its policy is eligible for purging.

In Studio, you can modify the behavior of the batch process by modifying the worksheet purge plugin.

#### See also

- “Worksheet Purge Plugin” on page 545

#### Enabling the Batch Process

In the base configuration, the Purge Rating Worksheets batch process is disabled. To enable this batch process, you must do the following in Studio:

1. In `config.xml`, set the `PurgeWorksheetsEnabled` parameter to `true`.
2. In `scheduler-config.xml`, to enable the batch process to run on a regular schedule, uncomment the `ProcessSchedule` entry for `PurgeWorksheets` and modify the schedule to meet your needs.

## Worksheet Extract Plugin

The `WorksheetExtractPlugin` plugin interface identifies which rating worksheet to extract and extracts the worksheet data to files.

PolicyCenter includes a built-in implementation of this plugin interface. See `PCWorksheetExtractPlugin` in the `gw.plugin.purge.impl` package.

#### Identifying Worksheets to Extract

PolicyCenter calls the `buildWorksheetCandidatesQuery` method to determine which worksheets to extract.

In `PCWorksheetExtractPlugin`, this method identifies `WorksheetContainer` objects attached to a `Policy` on which all jobs are closed. Closed jobs have a non-null `Job.CloseDate`. The method ignores `WorksheetContainer` objects that have already been extracted.

#### Extracting Worksheet Data

PolicyCenter calls the `extractWorksheets` method to extract the worksheet data into a file and flag the worksheet container for purging.

In `PCWorksheetExtractPlugin`, the worksheet data is extracted to XML and compressed into Gzip format. The format of the file name is:

`PolicyNumber_TermNumber_JobNumber_BranchPublicID.gz`

#### Can Purge Extracted Worksheets

PolicyCenter calls the `isExtracted` method. The method returns a value which indicates whether data has been extracted from a worksheet container.

In `PCWorksheetExtractPlugin`, this method retrieves the value of the `CanPurge` Boolean property on the worksheet container. The `CanPurge` flag is false by default. This flag is `true` if:

- Data has been extracted
- There is no data to extract

The Purge Worksheet Containers batch process uses the `CanPurge` flag.

## Worksheet Purge Plugin

The `WorksheetPurgePlugin` plugin interface configures how PolicyCenter purges rating worksheets.

PolicyCenter includes a built-in implementation of this plugin interface. See `PCWorksheetPurgePlugin` in the `gw.plugin.purge.impl` package.

The `buildWorksheetCandidatesQuery` method builds the list of worksheet containers that meet the requirements for the Worksheet Purge batch process to purge.

In `PCWorksheetPurgePlugin`, the `buildWorksheetCandidatesQuery` method finds worksheet containers that have the `CanPurge` property set to `true`, and whose jobs were completed more than `RatingWorksheetContainerAgeForPurging` days ago. The `RatingWorksheetContainerAgeForPurging` parameter is defined in `config.xml`.

PolicyCenter calls the `canPurgeWorksheetContainer` method to determine if the rating worksheet can be purged. In the base implementation, this method just returns the value of the `CanPurge` flag on the `WorksheetContainer`. You can include additional logic in this method.

## Logging Rate Routine Functions in a Rating Worksheet

In the default configuration, logging to the rating worksheet occurs automatically for each step in the rate routine.

You can add logging to the rating worksheet for a rate routine function by using methods in the `WorksheetLogger` class in the `gw.rating.worksheet` package.

In the default configuration, the `capPremiumByPercent` method in the `SharedRatingFunctions` class provides an example of how to use the `WorksheetLogger` methods. The `SharedRatingFunctions` class is in the `gw.rating.flow.util` package.

**Note:** In the default configuration, the personal auto and commercial property lines of business have logging for rating worksheets. For information about configuring rating worksheet for other lines of business, see “Configuring Rating Worksheets” on page 543.

### See also

- “Adding a New Rate Routine Function” on page 534

## Configuring Impact Testing

This topic describes how to configure impact testing. You can enable impact testing for a line of business and modify the functionality of impact testing. In addition, there are batch processes associated with impact testing.

- “Enabling Impact Testing for a Line of Business” on page 547
- “Modifying the Functionality of Impact Testing” on page 548
- “Work Queues for Impact Testing” on page 548
- “Configuring the Impact Testing Plugin” on page 548

### Impact Testing Warnings and Recommendations

Impact testing is designed to work in a test environment on a copy of production data. Impact testing is accessible only when the server is in development or test mode. Because impact testing affects system performance and creates test policy periods that persist in the database, impact testing is not accessible in production mode.

When you run impact testing, impact testing creates test policy periods in the database. In the test environment, observe the following warnings and recommendations:

- The work of users can interfere with impact testing results. During impact testing, Guidewire recommends that only the single user performing impact testing be logged into PolicyCenter.
- Disable integrations to other systems. If you run impact testing with integrations enabled, the integration can send test data to these production systems. For example, disable the free-text search integration that uses the search engine Solr. The same applies to other integrations such as an integration with a billing system. To avoid unnecessary costs, disable integrations to systems that charge access fees.

- While running impact testing batch processes, do not make changes in PolicyCenter that impact rating such as doing a policy change, canceling, or changing rate books. For example, the baseline creation batch process or the test period quote batch process is running. During this time, quoting test periods fails if a user changes or cancels a policy which has a baseline.
- Do not include expired policies in the impact testing dataset. Specify an **In Force On** date to filter out expired policies.
- Impact testing excludes archived policies.
- Stop the Job Expire batch process to prevent PolicyCenter from unexpectedly expiring policies during or between impact testing runs. To manage batch processes on a test server, press ALT + SHIFT + T to display the **Server Tools** page, then select **Batch Process Info** from the **Server Tools** tab.

**See also**

- “Impact Testing” on page 559 in the *Application Guide*
- “Working with Impact Testing” on page 593 in the *Application Guide*

## Enabling Impact Testing for a Line of Business

In the default configuration, impact testing is enabled for the personal auto and commercial property lines of business. To enable impact testing for another line of business, you must implement a getter for the **Coverable** property in the cost adapter for that line of business. For more information, see “Get Coverable” on page 556.

## Rating Renewals as Renewal Jobs in Impact Testing

In the default configuration, impact testing rates the baseline and test policy periods for all job types as submission jobs. You can configure impacting testing to rate a renewal as a renewal job rather than a submission job. For example, rate routines may include special steps for renewal jobs. Rate tables may contain factors based on job type.

In the default configuration, PolicyCenter prevents a renewal on a policy period which has an open renewal. If you make this configuration change, PolicyCenter can create a renewal on a policy period which has an open renewal. Making this configuration change impacts how PolicyCenter handles renewals in impact testing and in other areas of the product.

To rate renewals and renewal jobs in impact testing, change the **IPolicyPlugin** in Studio.

1. In Studio, navigate to **configuration** → **config** → **Plugins** → **registry** and open **IPolicyPlugin**.

2. Change **Gosu Class** from:

`gw.plugin.policy.impl.PolicyPlugin`

to:

`gw.plugin.rateflow.ImpactTestingPolicyPlugin`

The **ImpactTestingPolicyPlugin** plugin extends the **PolicyPlugin** plugin. The **ImpactTestingPolicyPlugin** plugin only contains an override of the **canStartRenewal** method.

**IMPORTANT** The **ImpactTestingPolicyPlugin** plugin is only intended for use in a development environment where you are performing impact testing. Do not use this plugin in a production environment. If you enable the **ImpactTestingPolicyPlugin** plugin, PolicyCenter allows the creation of a renewal on a policy period with an open renewal. In the default configuration, PolicyCenter prevents a renewal on a policy period that has an open renewal. Enabling the **ImpactTestingPolicyPlugin** changes how PolicyCenter handles renewals in impact testing and in other areas of the product.

## Modifying the Functionality of Impact Testing

In the PolicyCenter default configuration, impact testing creates the test periods by creating policy renewals. You can customize how impact testing creates the test periods. For example, you may want to create test periods in another way, such as by copying submissions or spinning off policies. You can also change how to select alternate rate books in impact testing.

You can make these changes by writing your own implementation of the impact testing plugin interface.

### See also

“Impact Testing Plugin” on page 553

## Work Queues for Impact Testing

In Guidewire Rating Management, impact testing uses the following work queues:

Work queue name	Code	Description
Impact Testing Test Case Preparation	ImpactTestingTestPrep	This work queue generates baseline policy periods on the selected policies when you click <b>Create Baselines</b> from the <b>Create Baseline</b> screen.
Impact Testing Test Case Run	ImpactTestingTestRun	This work queue generates test policy periods rated using the selected rate books when you click <b>Quote Test Periods</b> from the <b>Testing Periods</b> screen.
Impact Testing Export	ImpactTestingExport	This work queue exports test periods to an Excel file when you click <b>Create Excel Export File</b> on the <b>Impact Results</b> screen.

You can access impact testing in PolicyCenter by navigating to **Administration** → **Rating** → **Impact Testing**.

### See also

- “Batch Processing” on page 99 in the *System Administration Guide*

## Configuring the Impact Testing Plugin

You can use impact testing to see the impact that changing the rate book has on the policy premium.

The impact testing plugin (`IImpactTestingPlugin`) generates the baseline and test periods for impact testing. The plugin also gets and sets the rate books for quoting the test periods.

The built-in plugin implementation, `gw.plugin.rateflow.ImpactTestingPlugin.gs`, creates test periods by creating policy renewals. You can create your own implementation of this plugin. For example, you may want to create test periods in another way, such as by copying a submission or spinning off a policy. You can also change how the alternate rate books are selected.

For more information, see “Impact Testing Plugin” on page 553.

## Rating Management Plugins and Interfaces

This topic describes plugins related to Guidewire Rating Management features.

- “Rate Routine Plugin” on page 549
- “Impact Testing Plugin” on page 553
- “Adding Line-specific Cost Methods Using Cost Adapters” on page 555

## Rate Routine Plugin

PolicyCenter provides the `IRateRoutinePlugin` to let you modify the processing of rate routines in PolicyCenter. The default configuration of PolicyCenter uses a built-in implementation of the plugin, `RateRoutinePlugin.gs`. You can create your own implementation of this plugin. To edit the registry for `IRateRoutinePlugin`, In Studio, navigate to **configuration** → **config** → **Plugins** → **registry** and open `IRateRoutinePlugin.gwp`.

When PolicyCenter executes a rate routine, PolicyCenter calls the rate routine plugin (`IRateRoutinePlugin`). The rate routine plugin:

- Enable rating worksheets in the plugin.
- Sets up cost data wrappers that determine which properties on the cost appear when you select an instruction or operand in a rate routine step.

### See also

- “Rate Routines” on page 550 in the *Application Guide*
- “Configuring Rate Routines” on page 533

## Rate Routine Plugin Implementation

PolicyCenter includes a built-in implementation of the rate routine plugin. This implementation contains classes for the personal auto and commercial property lines of business. In Studio, the built-in plugin implementation is `RateRoutinePlugin.gs` in the `gw.plugin.rateflow` package. This plugin implementation is read-only. You can create your own implementation of this plugin to customize the rate routine plugin logic.

The `RateRoutinePlugin.gs` passes the processing to line-of-business-specific configuration classes, if a class exists. The `RateRoutinePlugin.gs` looks for the line-of-business configuration classes in the `gw.lob` package. To find these configuration classes, the plugin maps the policy line code to the product abbreviation as follows:

Policy Line Code	Product Abbreviation	Configuration Class
PersonalAutoLine	PA	<code>gw.lob.pa.rating.PARateRoutineConfig</code>
CPLine	CP	<code>gw.lob.cp.rating.CPRateRoutineConfig</code>

If the product abbreviation does not fit this mapping, the `makeAbbrevMap` and `makeConfigClassName` methods make adjustments to the mapping. In the `RateRoutinePlugin.gs`, these methods make an adjustment for the business auto line. If you need to make other adjustments, you must create your own version of the rate routine plugin.

The line-of-business configuration classes implement the `IRateRoutineConfig` interface.

## Enabling Rating Worksheets in the Rate Routine Plugin

This plugin has settings that enable the rate routine plugin to generate rating worksheets for a line of business. If rating worksheets are enabled for a line of business, the **Show Rating Worksheet** button appears on the **Quote** screen.

Enable rating worksheets in the plugin by making the following changes:

- Set the `WorksheetsEnabled` parameter to `true` in the `IRateRoutinePlugin` registry.
- In the `LOBRateRoutineConfig.gs` class, modify the `worksheetsEnabledForLine` override method to return `true`.

For example, you can turn on rating worksheets for a line of business by modifying the line-of-business configuration classes with an override to the `worksheetsEnabledForLine` method. For example, the following code in the `CPRateRoutineConfig` class enables rating worksheets for commercial property:

```
override function worksheetsEnabledForLine(linePatternCode : String) : boolean {
 return true
}
```

In the `RateRoutinePlugin` class, the `worksheetsEnabledForLine` method calls the line-specific implementation of `worksheetsEnabledForLine`.

#### See also

- “Configuring Rating Worksheets” on page 543 for additional steps to enable rating worksheet
- “Rating Worksheets” on page 558 in the *Application Guide*

## Cost Data

This plugin contains methods that set up cost data wrappers for the rate routine to use. These wrappers are used to define which properties on the cost appear when you select an instruction or operand in a rate routine step. The plugin must implement the `getCostDataWrapperType` and `makeCostDataWrapper` methods. In the built-in implementation, the rate routine plugin simply transfers processing to line-of-business configuration classes, if they exist.

The `makeCostDataWrapper` method returns a cost data wrapper for the given cost data. The wrappers in the sample implementation are:

- `CalcRoutineCostData`
- `CalcRoutineCostDataWithAmountOverride`
- `CalcRoutineCostDataWithPremiumCap`
- `CalcRoutineCostDataWithTermOverride`

These wrappers are in the `gw.rating.flow.domain` package.

## Rate Routine Plugin Methods

The rate routine plugin implementation must contain methods that do the following:

- “Set Parameters” on page 550
- “Get Cost Data Wrapper Type” on page 551
- “Make Cost Data Wrapper” on page 551
- “Enable Worksheets for a Policy Line” on page 551
- “Get Branching Fields” on page 552
- “Include Property” on page 552
- “Filter Irrelevant Items” on page 552

The built-in implementation of the rate routine plugin is `RateRoutinePlugin.gs` in the `gw.plugin.rateflow` package.

### Set Parameters

The `setParameters` method gets the values of the `WorksheetsEnabled` parameter defined in the plugin registry and sets the values on internal variables. The value defined in the plugin registry overrides the plugin setting. The method signature is:

```
override function setParameters(params : Map<Object, Object>)
```

### Get Cost Data Wrapper Type

The `getCostDataWrapper` method gets the cost data wrapper type. The method signature is:

```
override function getCostDataWrapperType(paramSet : CalcRoutineParameterSet) : IType
```

In the built-in implementation, `RateRoutinePlugin` class transfers processing to the method of the same name in the line-specific rate configuration class, such as `PARateRoutineConfig`.

In the line-specific configuration class, the `getCostDataWrapper` method gets the `IType` of the wrapper that will be used by rate routines for the given parameter set. The rate routine editor accesses a `CostData` instance using this wrapper type. The rate routine editor uses the wrapper to display the desired set of properties related to `CostData` in virtual form. For example, when entering an `Instruction` or `Operand` in a rate routine step in PolicyCenter, you can select a property on the cost such as `AdjustedRate`. The wrapper implementation maps `AdjustedRate` to the correct underlying `CostData` property: `StandardAdjRate`, `OverrideAdjRate`, or `ActualAdjRate`. For example, if the `CostData` supports overrides, this could be the override or standard property. If the `CostData` does not support overrides, it is almost certainly the standard property.

The parameter is:

paramSet	The parameter set for which to return the corresponding wrapper type
----------	----------------------------------------------------------------------

The method returns the `IType` corresponding to the instance that `makeCostDataWrapper` would return for the given `paramSet`.

### Make Cost Data Wrapper

The `makeCostDataWrapper` method makes the cost data wrapper. The method signature is:

```
override function makeCostDataWrapper(
 paramSet : CalcRoutineParameterSet,
 costData : CostDataBase,
 defaultRoundingLevel : Integer,
 defaultRoundingMode : RoundingMode) : ICostDataWrapper
```

In the built-in implementation, `RateRoutinePlugin.gs` transfers processing to the method of the same name in the line-specific rate configuration class, such as `PARateRoutineConfig.gs`.

In the line-specific configuration class, the `makeCostDataWrapper` method makes a wrapper for the given `CostData` instance.

The parameters are:

paramSet	The parameter set for which to return the corresponding wrapper
costData	An instance of the <code>CostData</code> subclass
defaultRoundingLevel	The default rounding level of the <code>CostData</code>
defaultRoundingMode	The default rounding mode of the <code>CostData</code>

The method returns a wrapper instance of the `IType` that `getCostDataWrapperType` returns for the `paramSet`.

### Enable Worksheets for a Policy Line

The `worksheetsEnabledForLine` method returns `true` if worksheets are enabled for a policy line. The method signature is:

```
override function worksheetsEnabledForLine(linePatternCode : String) : boolean
```

In the built-in implementation, `RateRoutinePlugin.gs` enables worksheets for a policy line if the `worksheetsEnabledForLine` method in the line-specific configuration class returns `true`.

The parameter is:

---

linePatternCode	Pattern code for a line of business
-----------------	-------------------------------------

---

The method returns `true` if worksheets are enabled for the line of business with the given pattern.

For more information, see “Adding an Implementation to a Plugins Registry Item” on page 110.

### Get Branching Fields

The property getter for `BranchingFields` returns an array which is an ordered list of zero or more branching fields which represent variant identifiers of a rate routine. The fields together with `Code` and `Version` on the `CalcRoutineDefinition` entity uniquely identify a rate routine. A rate routine and its variant identifiers have the same code. The signature is:

```
override property get BranchingFields() : IEntityPropertyInfo[]
```

In the default configuration, `jurisdiction` is an example of a rate routine variant identifier. In the PolicyCenter user interface, the `jurisdiction` variant is referred to as the *jurisdiction variant*.

If you add a new variant identifier, you must add the name of the variant identifier to the `fields` variable of the getter for the `BranchingFields` property.

For more information about adding a variant identifier, see “Configuring Variant Identifiers for a Rate Routine” on page 536.

### Include Property

The `includeProperty` method is a filter on which properties on `Parameters` the user can select in an `Instruction` or `Operand` of a rate routine step. Return `true` to make a property available. Return `false` to make a property unavailable. Return `null` to retain the default value of this property. It is recommended that this method end with a catch-all return `null`, so that any property which is not explicitly handled gets default behavior.

The signature is:

```
override function includeProperty(policyLinePatternCode : String, prop : IPropertyInfo) : Boolean
```

The parameters are:

---

policyLinePatternCode	The code for the policy line.
prop	The descriptor for the property to include.

---

This method operates during the traversal of the policy graph, and controls whether a reference from one class to another will be traversed. (In contrast, the `filterIrrelevantItems` method runs after the traversal of the policy graph.)

For example, `PolicyLocation` has a foreign key to an `AccountLocation`. In the built-in implementation, `PolicyLocation.AccountLocation` is not traversed. To change this behavior, you might add the condition:

```
if (prop.OwnersType.isAssignableFrom(entity.PolicyLocation) and prop.Name == "AccountLocation") {
 return true
}
```

### Filter Irrelevant Items

Similar to the `includeProperty` method, the `filterIrrelevantItems` method is a filter on which properties on `Parameters` the user can select in an `Instruction` or `Operand` of a rate routine step. However, this method runs after the traversal of the policy graph.

Irrelevant items are of the following types:

- Ignored paths – List of paths to filter out (defined in the `_ignoredPaths` variable). For example, the built-in implementation filters out `.EffectiveDate` and `.ExpirationDate`. See the plugin implementation for the complete list.
- Ignored types – List of object types to filter out (defined in the `_ignoredTypes` variable). For example, the built-in implementation filters out policy period and policy line pattern. See the plugin implementation for the complete list.
- Arrays or maps of types. You can optionally filter by policy line pattern code.

This filter runs after the traversal of the policy graph.

The signature is:

```
function filterIrrelevantItems(input : List<InScopeUsageBase>, policyLinePatternCode : String) :
List<InScopeUsageBase>
```

## Impact Testing Plugin

You can use impact testing to see the impact that changing the rate book has on the policy premium.

The impact testing plugin (`IImpactTestingPlugin`) generates the baseline and test periods for impact testing. The plugin also gets and sets the rate books for quoting the test periods.

PolicyCenter provides the `IImpactTestingPlugin` plugin interface so that you can customize impact testing logic in PolicyCenter. The default configuration of PolicyCenter uses a built-in implementation of the plugin, `ImpactTestingPlugin.gs`. This plugin implementation is read-only. You can implement your own instance of this plugin by copying and customizing this plugin implementation.

To edit the plugin registry for `ImpactTestingPlugin` in the Resources panel in PolicyCenter Studio, navigate to `configuration → config → Plugins → registry` and open `IImpactTestingPlugin.gwp`.

### See also

- “Impact Testing” on page 559 in the *Application Guide*
- “Configuring Impact Testing” on page 546

## Impact Testing Plugin Implementation

In Studio, the built-in implementation of this plugin is the `ImpactTestingPlugin.gs` class in the `gw.plugin.rateflow` package.

This plugin creates test periods by creating policy renewals. You can customize the implementation of this plugin directly. For example, you may want to create test periods in another way, such as by copying a submission or spinning off a policy. You can also change how the alternate rate books are selected.

## Impact Testing Plugin Methods

The impact testing plugin must implement methods that do the following:

- “Create Baseline Policy Period” on page 554
- “Create Test Policy Period” on page 554
- “Request a Quote on the Baseline Policy Period” on page 554
- “Request a Quote on the Test Policy Period” on page 554
- “Flag to Include the Policy Period” on page 555
- “Get Alternate Rate Books” on page 555
- “Set Alternate Rate Books” on page 555

### Create Baseline Policy Period

The `createBaselinePeriod` method creates a baseline policy period from the original bound policy period. The baseline policy period is rated using the selected rate books. For more information on rate book selection, see “Selecting the Rate Book Edition During Policy Rating” on page 554 in the *Application Guide*.

Impact testing compares the base policy period with the test policy period. The method signature is:

```
public PolicyPeriod createBaselinePeriod(PolicyPeriod original)
```

The parameter is:

Parameter	Description
original	The original bound policy period from which to create a baseline policy period.

This method returns the baseline policy period.

### Create Test Policy Period

The `createTestPeriod` method creates a test policy period from the original bound policy period. The test policy period is rated using the alternate rate books. The method signature is:

```
public PolicyPeriod createTestPeriod(PolicyPeriod original, PolicyPeriod baseline)
```

The parameters are:

Parameter	Description
original	The original bound policy period from which the baseline policy period was created.
baseline	The baseline policy period created by <code>createBaselinePeriod</code> .

This method can create the test policy period using either the original or baseline policy period, or a combination of both.

This method returns the test policy period.

### Request a Quote on the Baseline Policy Period

The `requestBaselineQuote` method request a quote on the baseline policy period. The quote rates the policy using the default rate books. The method signature is:

```
public void requestBaselineQuote(PolicyPeriod period)
```

The parameter is:

Parameter	Description
period	The baseline policy period

This method has no return value.

### Request a Quote on the Test Policy Period

The `requestTestQuote` method requests a quote on the test policy period. The quote rates the policy using the alternate rate books specified in “Set Alternate Rate Books” on page 555. The method signature is:

```
public void requestTestQuote(PolicyPeriod period)
```

The parameter is:

Parameter	Description
period	The test policy period.

This method has no return value.

#### Flag to Include the Policy Period

The `shouldIncludePeriod` flag indicates whether to include or exclude the policy period in impact testing. The method signature is:

```
public boolean shouldIncludePeriod(PolicyPeriod period)
```

The parameter is:

Parameter	Description
period	The policy period to be considered for inclusion or exclusion

If the return value is `true`, then include the policy period in the test policy periods.

#### Get Alternate Rate Books

The getter for the `AlternateRatebooks` property returns the rate books selected for rating the test policy periods. In the **Impact Testing**, → **Select Rate Books** screen in PolicyCenter, the alternate rate books appear under **Rate Books for Impact Testing**. The getter signature is:

```
property get AlternateRatebooks() : List<ImpactTestingRateBook>
```

This getter returns a list of alternate rate books.

#### Set Alternate Rate Books

The setter for the `AlternateRatebooks` property sets the alternate rate books for rating the test policy periods. You specify the alternate rate books in the `books` parameter. When querying for rate books, impact testing uses the alternate rate books instead of the default rate books. The setter signature is:

```
property set AlternateRatebooks(books : List<ImpactTestingRateBook>)
```

The parameter is:

Parameter	Description
books	A list of alternate rate books. Pass <code>null</code> to not override rate books.

This method has no return value.

## Adding Line-specific Cost Methods Using Cost Adapters

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. For example, the root cost entity for the personal auto line is `PACost`, and the cost adapter is `PACostAdapter.gs` in the `gw.lob.ba.financials` package.

The cost adapter interface contains methods related to creating transactions and reinsurance. In addition, the interface contains methods related to rating worksheets and impact testing in Guidewire Rating Management.

#### See also

- “Impact Testing” on page 559 in the *Application Guide*

## Cost Adapter Methods

The classes that implement the cost adapter interface must implement methods that do the following:

- “Create Transactions” on page 556
- “Get Reinsurable” on page 556
- “Get Coverable” on page 556

### Create Transactions

The `createTransaction` method creates a `Transaction` object in the given policy period branch that is appropriate for the `Cost` type. The method signature is:

```
Transaction createTransaction(PolicyPeriod branch)
```

### Get Reinsurable

The `getReinsurable` method gets the single reinsurable that is associated with this cost. The method signature is:

```
Reinsurable getReinsurable()
```

This method returns `null` if the cost does not represent a premium. For example, a cost for taxes or fees does not represent a premium.

### Get Coverable

Implement a getter for the `Coverable` property. The getter returns a coverable if there is a coverable associated with this cost. Impact testing uses this getter. The signature is:

```
override property get Coverable()
```

This getter returns the coverable associated with this cost or `null` if this cost is not associated with a coverable. This method can return `null` if impact testing is not enabled for this line of business.

# Configuring Reinsurance Management

This topic describes how to configure Reinsurance Management in PolicyCenter.

This topic includes:

- “Reinsurance Management Object Model” on page 557
- “Reinsurance Management Permissions” on page 563
- “Configuring Reinsurance Coverage Groups” on page 564
- “Reinsurance Management Underwriting Issues” on page 564
- “Implementing Gosu Methods for Reinsurance Management” on page 565

**See also**

- “Reinsurance Integration” on page 401 in the *Integration Guide*

## Reinsurance Management Object Model

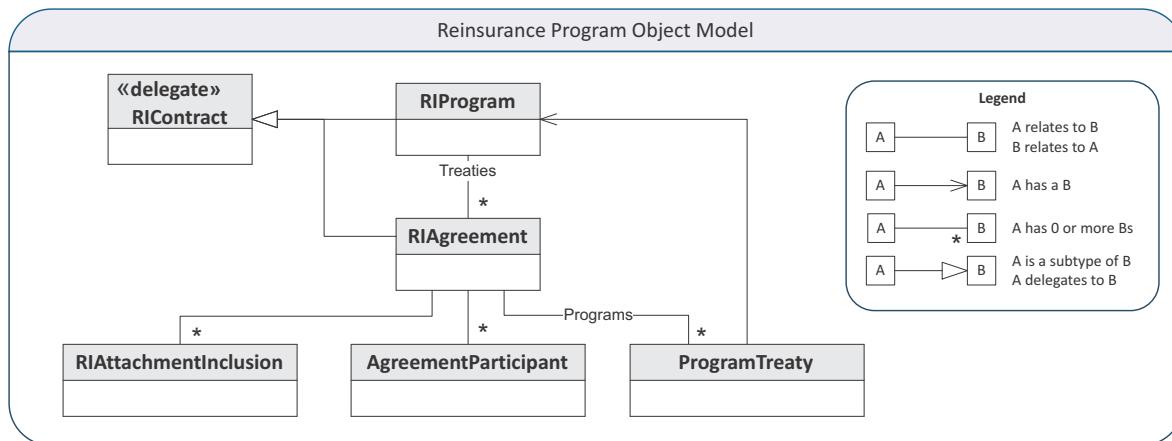
This topic describes the object model for Reinsurance Management and contains the following topics:

- “Reinsurance Program Object Model” on page 557
- “Reinsurance Agreement Object Model” on page 559
- “Reinsurance Coverage Group Object Model” on page 561
- “Policy Period Reinsurance Object Model” on page 563

### Reinsurance Program Object Model

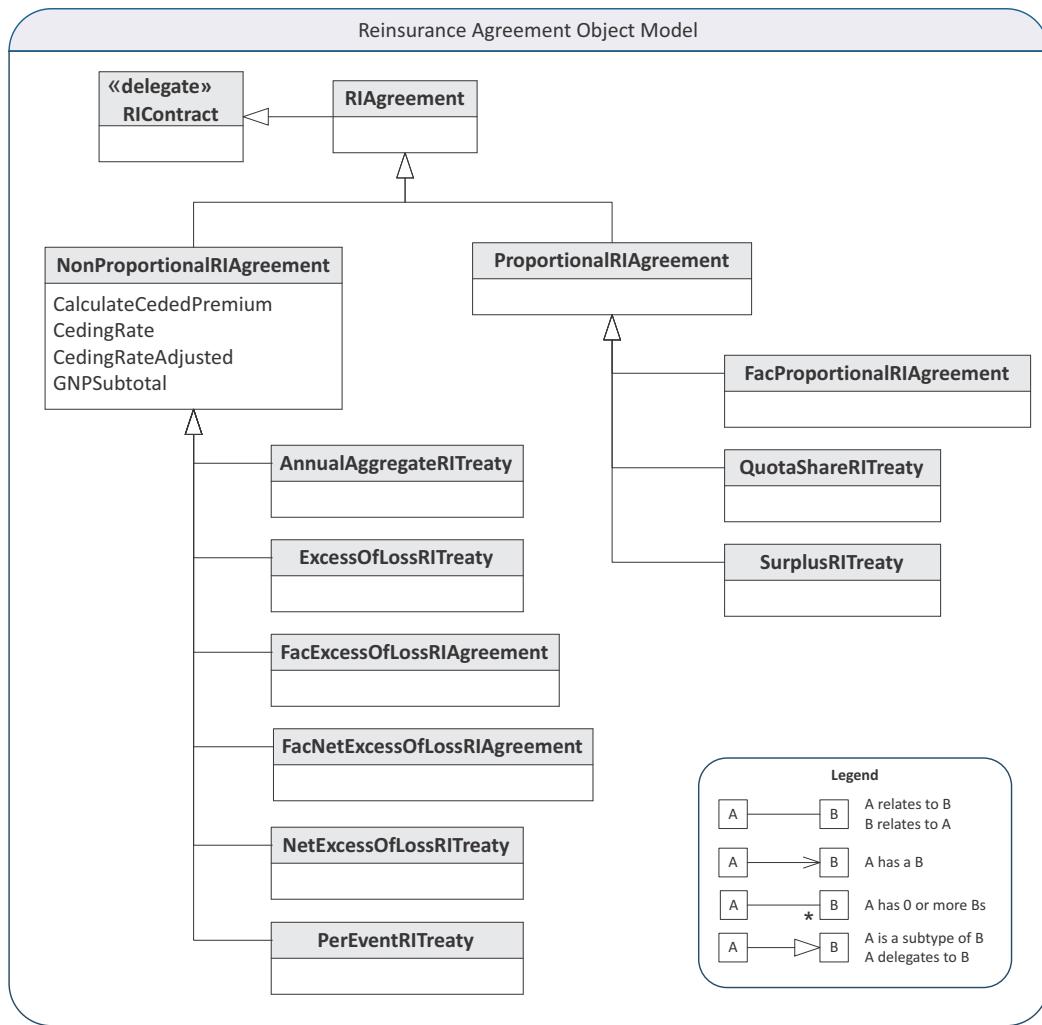
Reinsurance programs are represented by the RIProgram entity. Reinsurance programs contain one or more treaties.

Each reinsurance agreement is represented by the RIAGreement entity. A reinsurance agreement can be associated with one or more programs. You can access the programs that an RIAGreement is associated with through the Programs array. Each ProgramTreaty entity has a foreign key to an RIProgram.

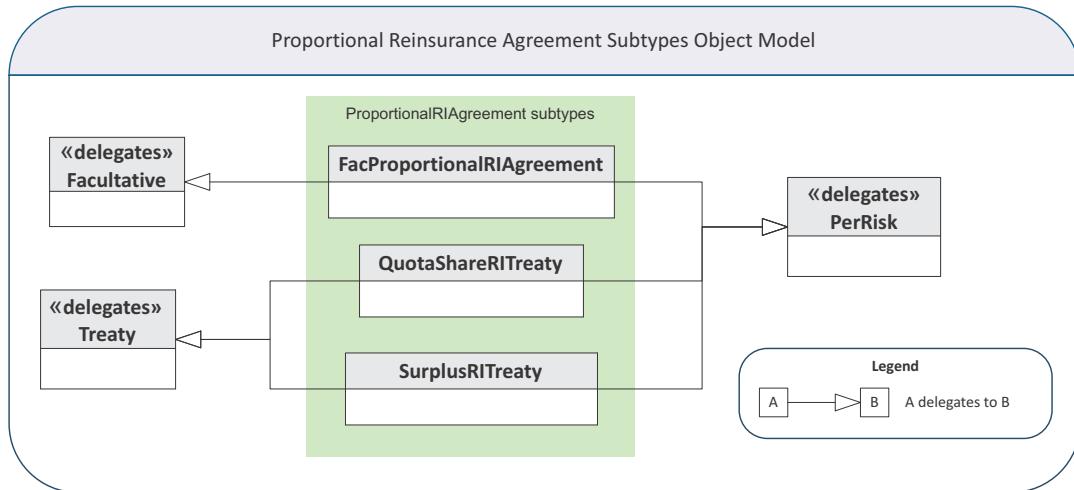


## Reinsurance Agreement Object Model

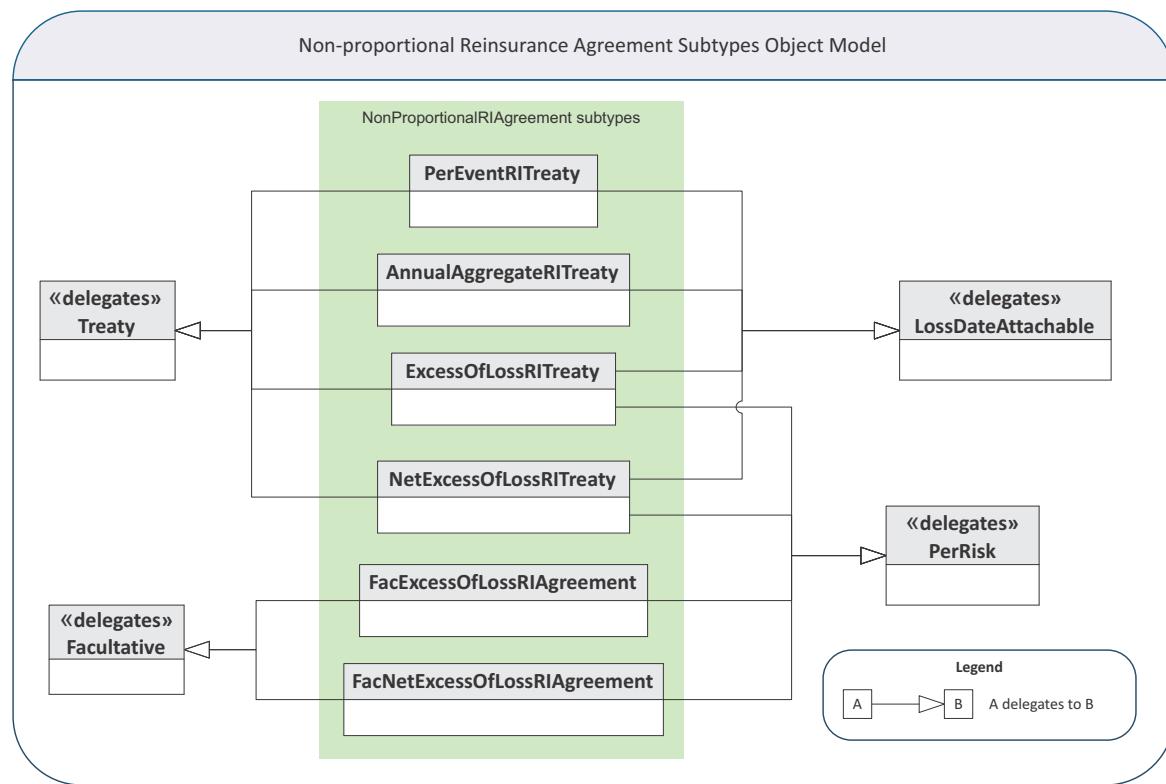
Reinsurance agreements are represented by the RI`Agreement` entity. The following illustration shows the subtypes of RI`Agreement`. Reinsurance agreements subtypes are proportional or non-proportional.



The following illustration shows the proportional reinsurance entities and their delegates.



The following illustration shows the non-proportional reinsurance entities and their delegates.



## Treaty Entities

All entities that represent treaties delegate to the Treaty entity. In the default configuration, the treaty entities are:

Proportional	Non-proportional
QuotaShareRITreaty	PerEventRITreaty
SurplusRITreaty	AnnualAggregateRITreaty ExcessOfLossRITreaty NetExcessOfLossRITreaty

## Facultative Agreement Entities

All entities that represent facultative agreements delegate to the Facultative entity. In the default configuration, the facultative agreement entities are:

Proportional	Non-proportional
FacProportionalRIAgreement	FacExcessOfLossRIAgreement FacNetExcessOfLossRIAgreement

## Loss Date Attachable Entities

All entities that represent loss date attachable delegate to the `LossDateAttachable` entity. In the default configuration, the loss date attachable entities are:

### Non-proportional

`PerEventRITreaty`  
`AnnualAggregateRITreaty`  
`ExcessOfLossRITreaty`  
`NetExcessOfLossRITreaty`

## Per Risk Entities

All entities that represent per risk agreements delegate to the `PerRisk` entity. In the default configuration, the per risk agreement entities are:

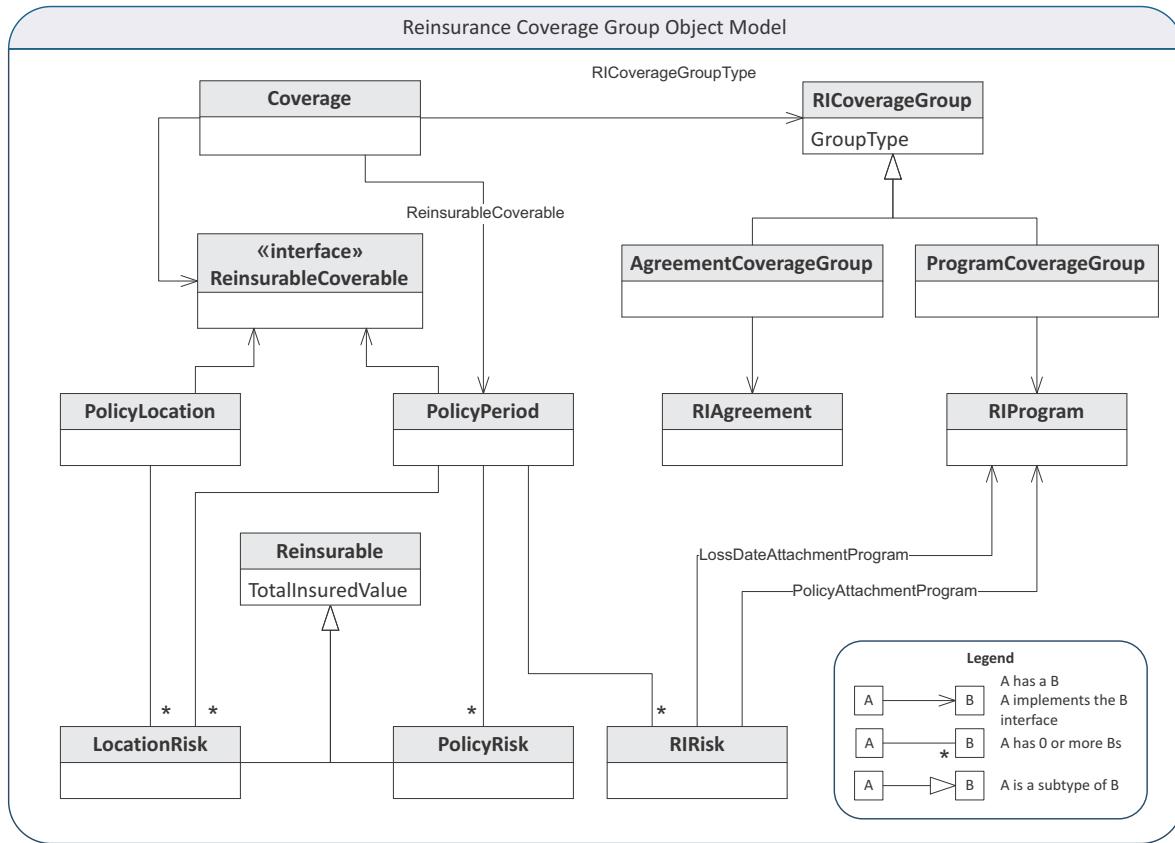
Proportional	Non-proportional
<code>FacProportionalRIAgreement</code>	<code>ExcessOfLossRITreaty</code>
<code>QuotaShareRITreaty</code>	<code>NetExcessOfLossRITreaty</code>
<code>SurplusRITreaty</code>	<code>FacExcessOfLossRIAgreement</code> <code>FacNetExcessOfLossRIAgreement</code>

## Reinsurance Coverage Group Object Model

The `Coverage` entity has two fields related to reinsurance. The `RICoverageGroup` field returns the `RICoverageGroupType` of the reinsurable. In the default configuration, `RICoverageGroupType` is a typelist with the following values:

- `Property`
- `Liability`
- `AutoPD`
- `AutoLiability`
- `WorkersComp`

The RICoverageGroupType field groups coverages of the same reinsurance coverage group into reinsurable risks. Two coverages of different reinsurance coverage groups cannot be grouped into the same reinsurable risk. This field is also used in reinsurance programs and reinsurance agreements to specify which reinsurance coverage groups these programs and agreements apply to.

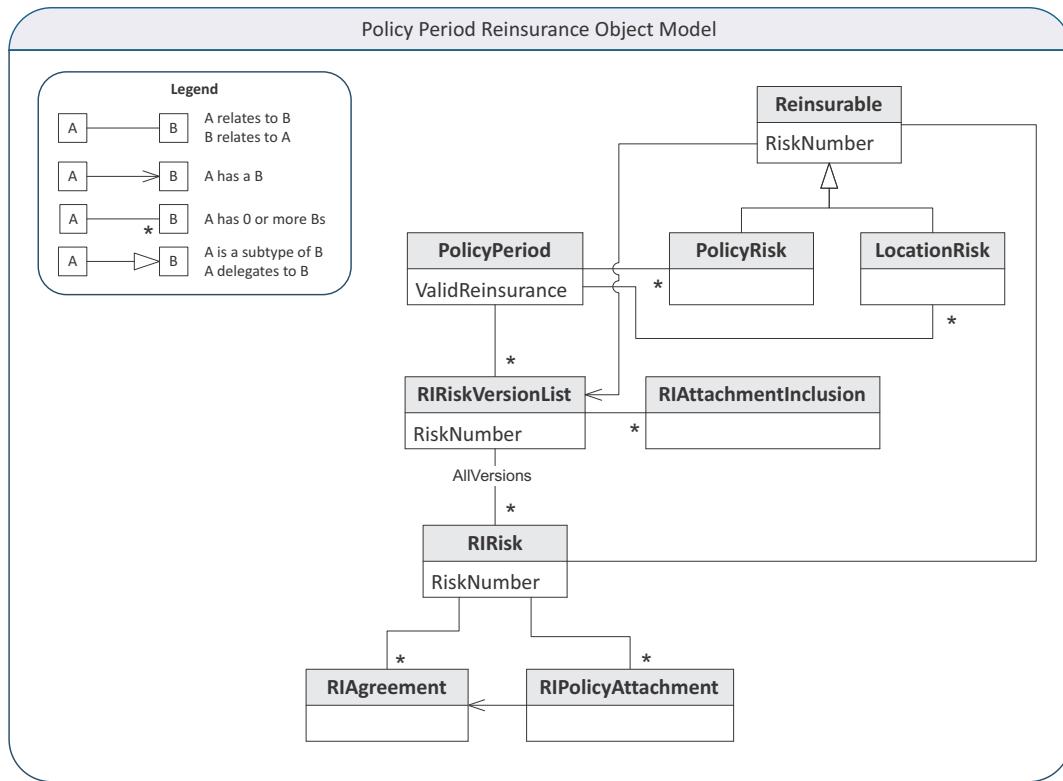


Coverages of same RICoverageGroup are not all grouped into a same reinsurable risk, but are divided further by what reinsurable coverable they apply to. In the default configuration, the reinsurable coverable entities are PolicyLocation and PolicyPeriod. The reinsurable risks associated with these coverables are LocationRisk and PolicyRisk, respectively. For example, if all property coverages for all buildings at the same policy location are grouped into a single reinsurable risk, the reinsurable coverable is the policy location.

Reinsurable coverables implement the ReinsurableCoverable interface. One ReinsurableCoverable can create multiple reinsurable risks depending upon how many RICoverageGroup entities the coverages under that ReinsurableCoverable have.

## Policy Period Reinsurance Object Model

This topic describes the reinsurance entities associated with jobs and policy periods.



The **RiskNumber** field is used to uniquely identify a risk in various entities in the object model. For example, the **RIRisk** and **PolicyRisk** entities have the same risk if they have the same **RiskNumber**.

The **RIRisk** entity delegates to the **SimpleEffDated** delegate which has **EffectiveDate** and **ExpirationDate** fields. Each instance of this entity represents a reinsurance risk for a period of time when it unchanged. The **RIRisk** represents a version of the risk.

The **RIRiskVersionList** entity is the container for all versions of a **RIRisk** created by a policy period. The version list has a link back to the policy period which created it.

The **RIPolicyAttachment** entity represents an attachment between a version of an **RIRisk** to an agreement. An **RIRisk** has an array to **RIAttachment**.

## Reinsurance Management Permissions

The following table lists the permissions related to Reinsurance Management:

Permission	Code	Description
Edit active RI programs	editreinsuranceactiveprogram	Permission to edit active reinsurance programs
Edit fac agreements	editreinsurancefacagreement	Permission to edit facultative agreements
Edit RI policies advanced	editreinsuranceforpolicyadvanced	Permission to edit advanced reinsurance fields for policies
Edit RI policies basic	editreinsuranceforpolicybasic	Permission to edit basic reinsurance fields for policies
Edit RI programs	editreinsuranceprogram	Permission to edit reinsurance programs

Permission	Code	Description
View RI policies	viewreinsuranceforpolicy	Permission to view reinsurance for policies
View RI programs	viewreinsuranceprogram	Permission to view reinsurance programs

## Configuring Reinsurance Coverage Groups

This topic describes how to configure reinsurance coverage groups in Product Designer.

### Configuring the Reinsurance Coverage Group on Individual Coverages

For each coverage, you can configure whether the coverage is part of a reinsurance coverage group.

1. In Product Designer, open a policy line and go to the **Coverages** page.
2. Select a coverage.
3. Go to the **Reinsurance** page for that coverage.
4. Select a **Reinsurance Coverage Group** from the drop-down list. You can also add a **Reinsurance Coverage Group Script**. Selecting a **Reinsurance Coverage Group** sets the **RICoverageGroup** on the **Coverage** entity.

### Disabling Reinsurance Management for a Line of Business

You can disable Reinsurance Management for a line of business. In Product Designer, open the policy line and make the following changes to each coverage on the **Coverages → Reinsurance** page:

- Set **Reinsurance Coverage Group** to <none selected>.
- Remove any **Reinsurance Coverage Group Script**.

If no coverages are linked to a coverage group, then no reinsurable risk will be created. Therefore, there will be no checking for reinsurance or linking to reinsurance agreements.

## Reinsurance Management Underwriting Issues

Reinsurance Management has one underwriting issue type.

### Reinsurance Net Retention Underwriting Issue

Reinsurance Management raises an **RINetRetention** underwriting issue if a policy period has reinsurable risks with insufficient reinsurance. PolicyCenter checks for this issue at quote release, bind, and issuance.

PolicyCenter raises an underwriting issue if the net retention is greater than the **Target Max Retention** in the **Reinsurance** screen of a policy file. In the base configuration, the **DefaultUnderwriterEvaluator.gs** class contains code to raise this underwriting issue.

If you integrate with an external reinsurance management system and have defined your own reinsurance data model, you will need to rewrite the Gosu code for raising this underwriting issue.

#### See also

- “Checking Set” on page 491 for more information about the **Reinsurance** checking set.

# Implementing Gosu Methods for Reinsurance Management

This topic describes the Gosu methods that you must implement for Reinsurance Management.

## Gosu Methods for Calculating the Total Insured Value

Each line of business calculates the total insured value.

In the default configuration, the `calculateTotalInsuredValue` method in `gw.api.policy.PolicyLineMethodsDefaultImpl` calculates the total insured value. For each coverage, this method calls the `getTIVForCoverage` method. Each line of business overrides the `getTIVForCoverage` method. For an example, see the `getTIVForCoverage` method in `gw.lob.ba.BAPolicyLineMethods.gs`.

If you implement a new policy line and do not use `PolicyLineMethodsDefaultImpl`, you must implement a `calculateTotalInsuredValue` method.

If the new policy line inherits from `PolicyLineMethodsDefaultImpl`, the policy line needs to implement the `getTIVForCoverage` method.





## chapter 41

# Handling High Volume Quote Requests

Some insurers need to handle high volumes of quote requests generated by external applications, such as the web sites of *comparative raters*. A comparative rater's web site enables its customer to view and choose between quotes from multiple insurers. Insurers who receive these requests must generate large volumes of quotes. The comparative rater's customer expects to see information quickly, therefore PolicyCenter must quickly generate the quote.

Most of these high volume requests do not result in a policy. Therefore, it is desirable to save the quotes in an external database rather than the PolicyCenter database. In general, this external database is partitioned to concurrently handle requests from multiple instances of PolicyCenter.

For these types of insurers, Guidewire recommends a quoting architecture and provides an API and plugin interface for generating quotes. To handle the high volume of quote requests, the architecture uses multiple instances of PolicyCenter. All instances leverage the same PolicyCenter version and configuration. To avoid database performance issues, the quote-only instances are not clustered. One or more *quote-only instances* process quotes directly from comparative raters. The architecture uses a separate PolicyCenter *system-of-record* (SOR) to complete submissions and issue policies.

**Note:** For efficient insertion and purging with an Oracle database, use composite range hash partitioning with the range based on a time interval, such as monthly. Partitioning by time enables you to easily purge old quotes. Use a similar mechanism if you are writing to another database.

**Handling lower volumes of quote requests** – Every PolicyCenter instance, even the SOR, includes the API which provide quotes that are stored outside the PolicyCenter instance. If you handle a lower volume of quote requests, you can use this API with the PolicyCenter SOR handling quote requests. Store the quotes outside the SOR database until they are bound.

**Keeping Quick Quotes outside the main database** – In a submission, you can choose to do a **Quick Quote** rather than a **Full Application**. In the base configuration, Quick Quote stores the quotes in the PolicyCenter SOR database. If your bind rate on these policies is low, you can modify Quick Quote to store the quote outside the SOR database until the quote is bound. You can do this by modifying the Quick Quote wizard to provide the quote and then call the Quoting Data plugin to write the quote to the quoting table.

This topic includes:

- “High Volume Quotes Overview” on page 568
- “High Volume Quotes Implementation Overview” on page 569
- “Quoting Processor Class” on page 571
- “Quoting Data Plugin” on page 571

# High Volume Quotes Overview

This topic describes a suggested way of handling high volume quote requests with PolicyCenter.

- “Generating the Quote” on page 568
- “Viewing the Quote” on page 568
- “Moving the Quote into the System of Record” on page 568
- “Other Considerations for Handling High Volume Quotes” on page 569

## Generating the Quote

High volume quote requests can be generated in the following way:

1. Incoming quote requests from comparative raters are translated into a set of PolicyCenter policy objects in a special quote-only instance of PolicyCenter. This instance only handles quote requests. Each request has a set of inputs that affect rating. Before sending the request, the comparative raters can make external calls to validate vehicle or address information.
2. The quote-only instance validates these objects. This validation may be customized for the comparative rater or can be the same as for other channels, such as call centers or portals.
3. To enhance the quote data, the quote-only instance can make external service calls, such as credit checks.
4. The quote-only instance rates the policy. Rating can be done by Guidewire Rating Management or another rating engine.
5. The rating engine returns pricing data and a reference number.
6. The quote-only instance stores both the original request and the completed quote in a Quote table, for example, located in an external quoting database. This quoting database has its own schema which does not necessarily follow the PolicyCenter object model, allowing for a different database storage strategy than the SOR database. The Quote table stores high-level quote details for identification and verification. The SOR can search this external quoting database without necessarily retrieving the full quote.
7. The quote-only instance sends the completed response back to the comparative rater.
8. The quote request/response data are immediately available for sending to a data warehouse for analytical purposes. If the quote results in a bound policy, the reference number can be used for auditing.

## Viewing the Quote

The comparative rater’s customer may not immediately decide to purchase a policy. Later, the customer can return to the comparative rater’s web site and view the quote again. The comparative rater retrieves the Quote from the external quoting database.

## Moving the Quote into the System of Record

When the comparative rater’s customer is ready to purchase a policy, the quote moves into the PolicyCenter SOR.

### From the Comparative Rater’s Web Site

If the comparative rater’s customer wants to purchase a policy while still on the comparative rater’s web site:

1. The customer is automatically navigated to the insurer’s web portal. The portal simultaneously makes a web service call to the PolicyCenter SOR to retrieve the Quote from the external quoting database using the reference number.

2. The SOR retrieves the Quote from the external quoting database and transforms it into a PolicyCenter submission. The SOR marks the record in the Quote table as converted to exclude it from further searches and multiple conversions.
3. The portal guides the user through additional questions and validation required to complete the submission.

#### From the Call Center

If the comparative rater's customer calls the call center to purchase a policy:

1. The call center consultant uses the PolicyCenter SOR screens to retrieve the Quote from the external quoting database using the reference number or other quote details.
2. The SOR retrieves the Quote from the external quoting database and transforms it into a PolicyCenter submission. The SOR marks the record in the Quote table as converted to exclude it from further searches and multiple conversions.
3. In PolicyCenter, the consultant guides the user through additional questions and validation required to complete the submission.

### Other Considerations for Handling High Volume Quotes

When setting up a systems to handle high volume quotes, also consider the following:

- All quote-only instances of PolicyCenter access the same Quote table in the external quoting database for storage and retrieval. Therefore, you can add additional instances as necessary to support peak loads. Each instance operates independently.
- The Quote table can be purged on a regular basis to keep the size manageable. All quotes which are past a specified validity date can be purged completely since the data warehouse has a record of the data.
- Maintenance on the PolicyCenter SOR will not impact the operation of the quote-only instances, and therefore facilitates 24/7 operation.

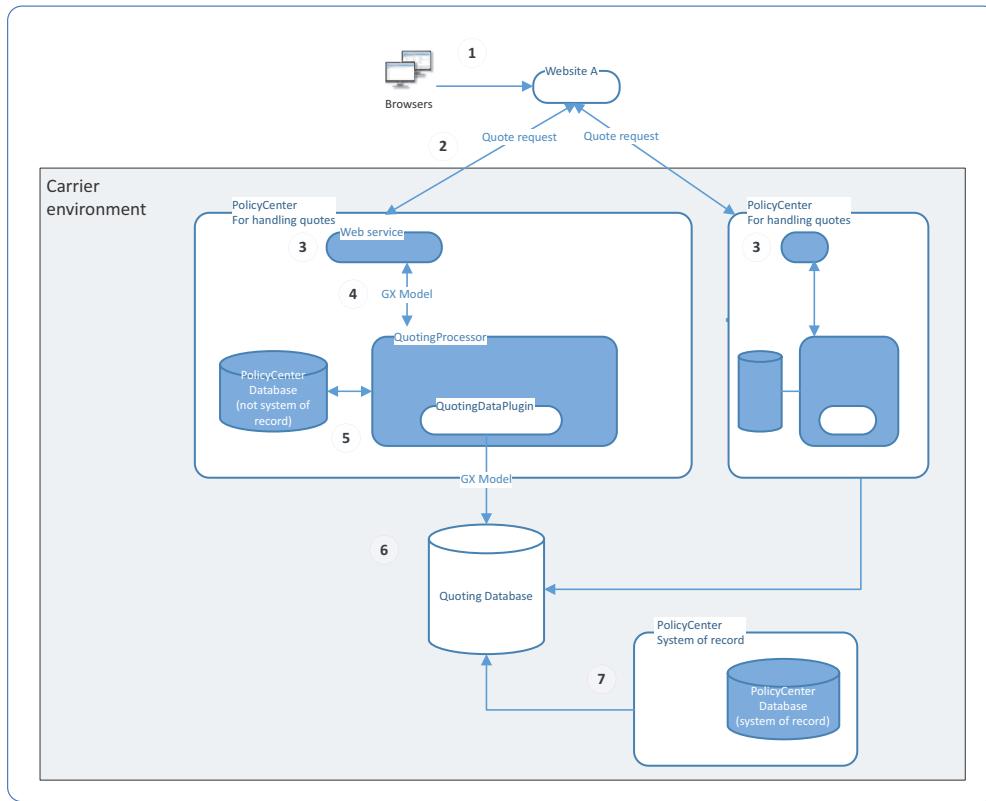
## High Volume Quotes Implementation Overview

To handle high volumes of quote requests, the `QuotingProcessor` class provides an API for generating quotes. This class calls methods on the Quoting Data (`QuotingDataPlugin`) plugin, which can save these quotes to an external database. Guidewire provides:

- APIs implemented in the `QuotingProcessor` class
- The Quoting Data plugin interface
- A sample plugin implementation in `StandAloneQuotingDataPlugin`. This implementation is a placeholder; it does not write to a database.

You must develop the Quoting Data plugin implementation that writes the GX model to your quoting database. You must also develop other pieces of this implementation, including web services, the GX model, and additional interactions with the quoting database. For example, you can write a custom web service that you publish from PolicyCenter. Your web service receives quote requests from external applications and sends them to the `QuotingProcessor` API. See “Web Services Introduction” on page 37 in the *Integration Guide*.

The following illustration presents a simplified model of how an insurer can use this API and plugin to handle quote requests.



1. Website A is a comparative rater. Their web site enables potential customers to view policy quotes gathered from multiple insurers. A potential customer on Website A enters basic policy information and requests quotes. One of these insurers uses PolicyCenter to handle these requests.
2. Website A sends the policy information to multiple insurers. One of these requests goes to a insurer with PolicyCenter.  
The insurer has multiple instances of PolicyCenter to handle these requests.
3. Your web service formats the request. The base configuration does not contain an example of this type of web service. The web service verifies the data, then formats the data in the GX model of a PolicyPeriod. See “The Guidewire XML (GX) Modeler” on page 304 in the *Gosu Reference Guide*.
4. Your web service then calls the `quoteSubmission` method in the Quoting Processor API (`gw.quoting.QuotProcessor`) to generate a quote.
5. The `quoteSubmission` method calls the Quoting Data plugin (`QuotingDataPlugin`) to obtain an account. Then this method generates a policy period and a quote. This method returns the quote and a quote identifier. Generating the policy period and quote access the PolicyCenter database attached to the PolicyCenter instance for handling quotes. This database is not a system of record. In this proposed implementation, neither the policy period nor the quote are committed to the PolicyCenter database. The external quoting database, described in the next step, is optimized for storing the quoting information.
6. The `quoteSubmission` method calls the Quoting Data plugin to save this quote to the external quoting database. The quoting database is partitioned so that it can concurrently handle requests from multiple instances of PolicyCenter.

7. If the potential customer decides to purchase the insurer's policy, then the PolicyCenter system of record retrieves the GX model of the quote from the external quoting database. The PolicyCenter SOR uses the quote identifier to retrieve the GX model. You implement this step.

Optionally, the Customer Service Representative can display the quote on the screen and make changes. The policy can be bound and saved to the PolicyCenter database.

## Quoting Processor Class

The quoting processor class (`gw.quoting.QuotingProcessor`) provides the `quoteSubmission` method to generate a policy and obtain a quote. PolicyCenter requires an account and the producer code associated with the account to produce a quote. This method calls the Quoting Data plugin (`QuotingDataPlugin`) to retrieve an account. The `quoteSubmission` method queries the PolicyCenter database to find the producer code associated with the account.

The input parameters to `quoteSubmission` are:

- `productCode` – The product code, such as `PersonalAuto` or `WorkersComp`. This is the `Code` defined in `Product Model` → `Products` in Product Designer.
- `policyPeriodData` – The policy period data as a Gosu XML object. This is the data with which to populate the policy period. The fully qualified name of the policy period type is:  
`gw.webservice.pc.pc800.gxmodel.quotingsolicyperiodmodel.PolicyPeriod`

The `quoteSubmission` method sends the quoting data to the Quoting Data plugin. The plugin saves the quoting data to an external database. The `quoteSubmission` method does not commit the policy period to the PolicyCenter database.

The `quoteSubmission` method returns a `QuoteData` object. This object contains a quote identifier and the quoting data formatted as a policy period in GX model format.

## Logging

The quoting processor class logs messages about quoting jobs (policy transactions) to the `PCLoggerCategory.QUOTING` category.

## Quoting Data Plugin

The Quoting Data plugin (`QuotingDataPlugin`) interface provides methods for saving quoting data to an external database. The built-in implementation of this plugin is `gw.plugin.quoting.StandAloneQuotingDataPlugin`. This plugin is for demonstration purposes only. You can write your own implementation of this plugin interface to communicate with your external database. For more information about plugins, see “Plugin Overview” on page 123 in the *Integration Guide*.

### Get Account

The `getAccount` method of the Quoting Data Plugin returns the account associated with the input parameter, `requestData`. The method signature is:

```
getAccount(requestData : Object) : Account
```

When the `QuotingProcessor` class calls `getAccount`, the `requestData` is policy period data in GX model format. To use the `requestData` argument, downcast it from `Object` to the appropriate GX model format. For example `gw.webservice.pc.pc800.gxmodel.quotingsolicyperiodmodel.PolicyPeriod` (the policy period type fully qualified name).

Your implementation of this method can use the `requestData` parameter to retrieve a PolicyCenter account. If you always use a single PolicyCenter account for these types of quotes, ignore the `requestData` argument and return that account. For example, you might use a single account if rating does not depend upon properties of the account. Using this single account avoids creating accounts for each request, since few of these accounts will be associated with full submissions. Similarly, if rating depends upon only a few account properties, the Quoting Data Plugin implementation can have one account per unique combination of properties.

## Send Quoting Data

The `sendQuotingData` method of the Quoting Data Plugin sends the quoting data to an external database. The return value of this method is a unique identifier for the quoting data. The unique identifier can be used later to retrieve the quote and bind a policy in the PolicyCenter system of record. The method signature is:

```
sendQuotingData(data : String) : Object
```

The plugin implementation can return any type of object that you require for the unique identifier. For example, the built-in implementation returns an `Integer`.

# Configuring Multicurrency

This topic describes how to configure multicurrency.

This topic includes:

- “Configuring PolicyCenter for a Single Currency” on page 573
- “Creating Multicurrency Policy Lines” on page 574
- “Configuring the Coverage Currency” on page 576
- “Configuring the Settlement Currency” on page 577
- “Configuring Multicurrency and Reinsurance” on page 579
- “Configuring Multicurrency and Rating” on page 580
- “Configuring Underwriting Authority and Multicurrency” on page 582
- “Implementing an Exchange Rate Service” on page 583
- “Enabling Multicurrency Integration” on page 584

**See also**

- “Multicurrency Features” on page 523 in the *Application Guide*
- “Configuring Currencies” on page 109 in the *Globalization Guide*

## Configuring PolicyCenter for a Single Currency

When you run PolicyCenter with `MultiCurrencyDisplayMode` disabled, currency fields are not visible in the user interface. Examples of currency fields in the user interface are currency selectors on screens and currency columns in tables.

In the base configuration, PolicyCenter runs with `MultiCurrencyDisplayMode` disabled.

To run PolicyCenter with `MultiCurrencyDisplayMode` disabled, set the following parameters in `config.xml`:

Parameter	Value	Description
<code>MulticurrencyDisplayMode</code>	<code>SINGLE</code>	The multicurrency display mode. In the base configuration, this parameter is set to <code>SINGLE</code> .
<code>DefaultApplicationCurrency</code>	A typecode from the Currency type-list.	The default currency for the application. Set to the currency of your choice.

#### See also

- “`MultiCurrencyDisplayMode`” on page 59
- “`DefaultApplicationCurrency`” on page 58

## Creating Multicurrency Policy Lines

This topic describes how to create policy lines that support more than one coverage currency. The tasks include:

- Identifying the coverage currencies that the line supports.
- Ensuring that the required foreign exchange conversions are available (not described in this topic)
- Adding appropriate coverage options for each of the currencies
- Managing rating in each of the currencies (not described in this topic)
- Enabling producers to write business in various settlement currencies

The examples use the Wright Construction account in the small sample data set. The examples assume that Wright Construction is based in the United States. The carrier provides a commercial property policy that insures buildings and locations in the United States and France. The following table contains currency information for these countries.

Country	Currency	Currency symbol	ISO 4217 Code
United States	U.S. dollar	\$	USD
France	Euro	€	EUR

---

**IMPORTANT** The `MultiCurrencyDisplayMode` parameter setting is permanent. After you change the value of `MultiCurrencyDisplayMode` to `MULTIPLE` and then start the server, you cannot change the value back to `SINGLE` again. The `MultiCurrencyDisplayMode` parameter is in `config.xml`.

---

## Adding Coverage Currencies to a Policy Line

This topic provides example of how to manage multiple coverage currencies for a given coverage. In these step-by-step instructions, you add the euro as a coverage currency for the commercial property line and add euro values to a money coverage term.

### Create workspace and change list

- In Product Designer, create a workspace named “Multicurrency”.

2. In Product Designer, create a change list named “Multicurrency Coverage Currency” that uses the workspace.

#### To add a currency to a coverage

1. In Product Designer, navigate to **Product Model** → **Policy Lines** → **Commercial Property Line**.
2. Under **Available Coverage Currencies**, click **Add** and select **EUR** from the drop-down list.  
Product Designer displays both **EUR** and the existing coverage currency as choices for coverages.
3. Go to **Coverages** and select **Building Coverage**.
4. Go to **Terms** and select the **Limit** coverage term.  
The **Value Type** of this coverage term is **Money**.
5. Click to **Add** a currency.
6. Select **EUR** and enter “75,000”, for example, as a default value. Optionally enter minimum and maximum values such as “25,000” and “100,000”.
7. Go to **Terms** → **Deductible**.  
The **Value Type** of this coverage term is **Money**. This is a option coverage term which presents a list of values.
8. Go to **Options**.
9. Add options in euros, such as 250, 500, and 750. Prefix the **Code** with **EUR** to distinguish the euro options, for example **EUR250**.
10. Commit the change list and synchronize the product model.

#### To check your work

1. Log in to PolicyCenter as a producer such as **aapplegate**.
2. Go to the Wright Construction account.
3. Create a Commercial Property submission.
4. On the **Policy Info** screen, verify that **USD** and **EUR** appear in the **Preferred Currency** → **Coverage** drop-down list and select **EUR**.  
The preferred coverage currency will be the default currency for coverables on the policy.
5. Advance to the **Buildings and Locations** screen.
6. Add a building to the primary location and fill in any required fields on the **Building Information** screen.  
Verify that the **Coverages** in drop-down list includes **USD** and **EUR**, and the preferred coverage currency, **EUR**, is selected.
7. Fill in required fields on the **Details** tab.
8. Go to the **Coverages** tab.  
Notice that the **Limit** is set to the default value for **EUR**. The **Deductible** drop-down list contains the **EUR** options you defined in Product Designer.
9. Deselect all coverages except for **Building Coverage**.
10. Advance to the **Policy Review** screen.  
Notice that for the building, the **Coverages** in are in euros.
11. Quote the policy.  
The quote is in U.S. dollars because that is the settlement currency.

## Changing the Settlement Currency

The preferred settlement currency, `PreferredSettlementCurrency`, is a field on the contact, account, and policy period entities.

In a job, the default settlement currency is the settlement currency on the account. In a job, you can pick a settlement currency from the `Preferred Currency` → `Settlement` drop-down list on the `Policy Info` screen. The list of `Settlement` currencies is populated from the `Currency` typelist.

A producer can quote a policy in any configured currency. However, when you bind or issue a policy, the producer of record must have the authority to bind the policy in the settlement currency. In the base configuration, the user interface enables you to select only one currency per producer code. You can configure PolicyCenter to allow more than one currency per producer code. You can view producer codes by navigating in PolicyCenter to the `Administration` → `Producer Codes` screen.

## Configuring the Coverage Currency

In the base configuration, the coverage currency appears in various places in accounts and policies. In the base configuration, accounts, policy periods, and coverables have coverage currency properties. Coverages and clauses have a currency property.

In the base configuration, you set the coverage currency on a coverable. PolicyCenter sets the currency for each coverage on the coverable to the same currency as the coverable. You can modify this behavior through configuration.

### Coverage Currency in Accounts

Accounts have a preferred coverage currency. You can view or set this value on the `Account Summary` screen. The preferred coverage currency on account is in the `Account.PreferredCoverageCurrency` property. In the base configuration, PolicyCenter initially sets the account's preferred coverage currency to the contact's preferred settlement currency, `Contact.PreferredSettlementCurrency`. You can change this behavior through configuration.

In the base configuration, all contact objects have a preferred coverage currency even if PolicyCenter is configured for a single currency. (When running PolicyCenter with `MultiCurrencyDisplayMode` disabled, the currency does not appear in the user interface.) However, if a contact does not have a preferred coverage currency, then PolicyCenter sets the account's preferred coverage currency to the default system currency. A properly configured system always specifies the currency and does not rely on this fail-safe behavior.

### Coverage Currency in Policy Periods

Policy periods have a preferred coverage currency which PolicyCenter initializes when initializing the policy term. The preferred coverage currency on policy periods is in the `PolicyPeriod.PreferredCoverageCurrency` property. In the base configuration, this property is set to the account's preferred coverage currency property, `Account.PreferredCoverageCurrency`, if it exists. This is implemented in the `initializeCurrencies` method in the `PolicyPeriodPlugin` class in the `gw.plugin.policyperiod.impl` package.

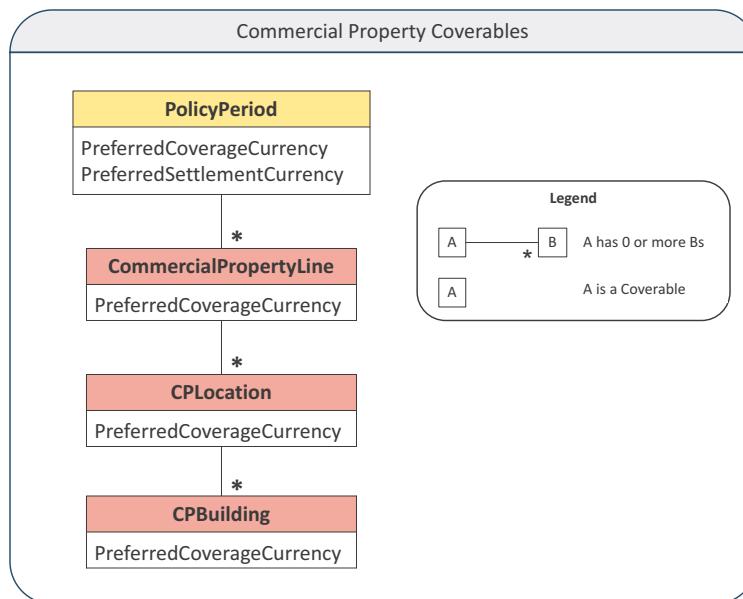
In the base configuration, all accounts have a preferred coverage currency even if PolicyCenter is configured for a single currency. However, if an account does not have this value, PolicyCenter sets the policy period's preferred coverage currency to the default system currency. A properly configured system always specifies the currency and does not rely on this fail-safe behavior.

To get the default preferred coverage currency for the specified policy period, you can use the `LookupDefaultPreferredCoverageCurrency` method in the `gw.job.PCPolicyCurrencyUtil` Gosu class.

## Coverage Currency in Coverages and Clauses

Coverages, exclusions, and policy conditions include a currency which is the currency for the clause. The `Currency` property on the `Coverage`, `Exclusion`, and `PolicyCondition` entities stores the currency. In the base configuration, all coverages on the same coverable have the same currency. Therefore, the user interface does not provide a way to change the currency on a specific clause. The user interface lets you change the currency on the coverable, and that currency is propagated to all coverages on the coverable.

A coverable inherits its default currency from the owning coverable or policy period. In Commercial Property for example, a new building (`CPBuilding`) automatically gets the preferred coverage currency, `PreferredCoverageCurrency`, of its parent coverable, `CPLocation`.



## Coverage Currency in Coverables

Coverables have a preferred coverage currency which PolicyCenter uses to synchronize all coverages associated with a coverable. The preferred coverage currency on coverables is in the `Coverable.PreferredCoverageCurrency` property. After updating the preferred coverage currency on a coverable, PolicyCenter synchronizes the product model. The `synchronizeCurrencies` method in the `gw.web.policy.CoverableCoverageCurrencySynchronizer` Gosu class synchronizes the product model.

## Configuring the Settlement Currency

This topic describes Gosu classes and object related to the settlement currency. The settlement currency is used in billing. In the base configuration, the contact, account, policy period objects have a settlement currency property.

### Settlement Currency in Contacts

Contacts have a preferred settlement currency. The preferred settlement currency on contact is in the `Contact.PreferredSettlementCurrency` property. PolicyCenter initializes the preferred settlement currency in the `gw.account.AccountBaseEnhancement` class.

In the base configuration, the preferred settlement currency is based on the primary address of the contract. For example, if the primary address is in California, which is in the United States, then PolicyCenter sets the currency to U.S. dollars. The `createAccountForContact` method sets the preferred settlement currency.

In the base configuration, the jurisdiction on the address maps to a country, and each country has a currency. However, if for some reason an address does not map to a currency, the `createAccountForContact` method initializes this property to the system default currency. A properly configured system always specifies the currency and does not rely on this fail-safe behavior.

The `ContactCurrencyInitializer` class contains methods for editing the currency for a contact. PCF files that enable a user to edit the currency for a contact use these methods. The `ContactCurrencyInputSet` PCF file uses this class to set the **Preferred Currency** for a contact. You can see an example of this **Preferred Currency** field when you view a contact on the **Contact File Details** screen.

The **Account Holder Info** screen uses the preferred settlement currency on the contact to aggregate policy, claim, and billing metrics and display them in the settlement currency. The contact's preferred settlement currency becomes the default settlement currency for an account created from that contact.

## Settlement Currency in Accounts

Accounts have a preferred settlement currency. The preferred settlement currency on account is in the `Account.PreferredSettlementCurrency` property. To create a new account with the given contact as the account holder, you can use the `createAccountForContact` method in `gw.account.AccountBaseEnhancement`. In the base configuration, this method initializes the account's preferred settlement currency based on the contact's primary address. The code assumes that if the coverage currency is set, the settlement currency is also set.

In the base configuration, the jurisdiction on the address maps to a country, and each country has a currency. For the rare jurisdiction that does not have a country, you can specify the currency directly for that jurisdiction. Map the jurisdiction to currency in the definition of the jurisdiction and country typekeys. For an example, see the typecode for Guam in the `Jurisdiction.ttx` typelist extension. However, if for some reason an address does not map to a currency, the `createAccountForContact` method initializes this property to the system default currency. A properly configured system always specifies the currency and does not rely on this fail-safe behavior.

When the account is initially created, the country of the account's address determines the preferred settlement currency. Methods in the `gw.web.financials.AccountCurrencyInitializer` class determine the currency.

The `AccountCurrencyInputSet` PCF file displays the account's settlement currency.

The `AccountFile_Claims` and `SubmissionManager` PCF files use the settlement currency to determine the currency on the **Account File Claims** and **Submission Manager** screens. These files use the settlement currency for computing and displaying the **Total Incurred** and **Total Cost** fields, respectively.

## Settlement Currency in Policy Periods

Policy periods have a preferred settlement currency. The preferred settlement currency is in the `PolicyPeriod.PreferredSettlementCurrency` property. PolicyCenter initializes this field when initializing the policy term. You can modify the initialization by overriding the `initializeCurrencies` method in the `PolicyPeriodPlugin` plugin interface.

The `lookupDefaultPreferredSettlementCurrency` method in the `gw.job.PCPolicyCurrencyUtil` class returns the default preferred settlement currency for the specified policy period.

When you create a policy in the user interface, you set the policy period's settlement currency (**Preferred Currency** → **Settlement**) on the **Policy Info** screen.

## Configuring Multicurrency and Reinsurance

PolicyCenter attaches a reinsurance program to a risk in a policy if the *total insured value/sum insured* (TIV/SI) for that risk has the same currency as the risk. In the base configuration, the TIV/SI currency is the currency associated with the jurisdiction of the risk. Through configuration, you can modify how PolicyCenter chooses the TIV/SI currency.

The reinsurance currency is independent of the coverage and settlement currencies. In the base configuration, PolicyCenter sets the reinsurance currency to the currency specified for the coverable's jurisdiction in the `ReinsuranceConfigPlugin` plugin. This plugin implements the `IReinsuranceConfigPlugin` plugin interface. The plugin calculates the TIV/SI based on the coverage currency. If the coverage currency differs from the reinsurance currency, the plugin converts the TIV/SI to the reinsurance currency. The `Reinsurable` object stores the TIV/SI (`TotalInsuredValue`) and reinsurance currency (`ReinsuranceCurrency`). The `Reinsurable` is then used to create the reinsurable risk (`RIRisk`) objects. If an appropriate reinsurance program is available, PolicyCenter attaches the program to the reinsurable risk. The program has the same currency as the reinsurable risk.

In the `ReinsuranceConfigPlugin` plugin, the `getReinsuranceCurrency` method returns the currency that the `RIRisk` uses for TIV/SI. To change how PolicyCenter selects the reinsurance currency, you can override this method.

The input parameter to the `getReinsuranceCurrency` method is a list of all the coverages associated with a single `ReinsurableCoverable`. The `PolicyPeriodBaseEnhancement` Gosu class calls this method when creating `Reinsurables`. In the base configuration, the method gets the jurisdiction specified in the `Coverable.CoverableState` property and looks up the currency associated with that jurisdiction. That mapping is returned by the `RegionCurrencyMappingUtil` class and configured through the `Country`, `Jurisdiction`, `State`, and `Currency` typelists.

The reinsurance program and reinsurance contract entities, `RIProgram` and `RIAgreement` respectively, delegate to the reinsurance contract entity, `RIContact`. The `RIContact` entity has a non-nullable currency property, `Currency`, that the `RIProgram` and `RIAgreement` entities implement. The program and agreements associated with the program must have the same currency.

When PolicyCenter attaches an `RIProgram` or an `RIAgreement` to an `RIRisk`, the `Currency` properties must match.

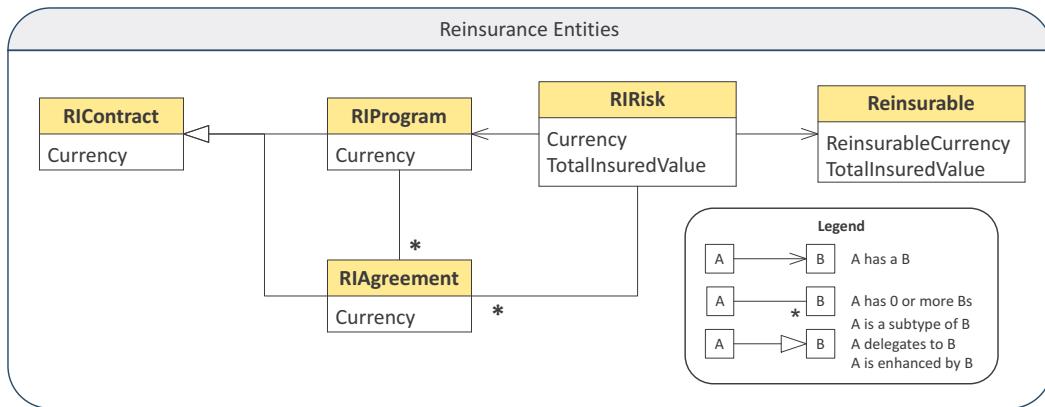
The Premium Ceding batch process runs and calls the `PCCedingCalculator` class to process premium transactions. This class does the conversion from the reinsurance currency to the settlement currency.

### See also

- “Reinsurance Configuration Plugin” on page 411 in the *Integration Guide*

## Reinsurance Objects and Multicurrency

The following illustration shows some of the multicurrency properties on reinsurance objects.



## Configuring Multicurrency and Rating

In the base configuration, the rating engine creates cost data in the currency of the associated coverage.

For costs not associated with a coverage, the rating engine uses the `PreferredCoverageCurrency` on the policy period associated with the line being rated.

The `gw.rating.AbstractRatingEngineBase` class defines a `TaxRatingCurrency` property and sets it to the `PreferredSettlementCurrency` on the policy period associated with the line being rated. The `AbstractRatingEngineBase` class is a read-only file.

### See also

- “Multicurrency and Rating” on page 525 in the *Application Guide*

## Costs and Multicurrency

The rating engine rates the coverables in the coverage currency and returns cost data to PolicyCenter. PolicyCenter converts the cost data into Cost objects. PolicyCenter stores the currency and the amount on `Amount` properties on the Cost, for example `Cost.ActualAmount`. The `Amount` properties are of `MonetaryAmount` type which has properties for an amount and a currency. If necessary, PolicyCenter converts the amounts to the settlement currency. The rating engine then stores the amounts in duplicate billing amount properties (`AmountBilling`) on the Cost entity, for example `Cost.ActualAmountBilling`. If the coverage currency differs from the settlement currency, the rating engine also stores a pointer to the exchange rate. The `Cost.CurrencyChanged` Boolean property is set to `true` if the PolicyCenter converted the currency on the cost.

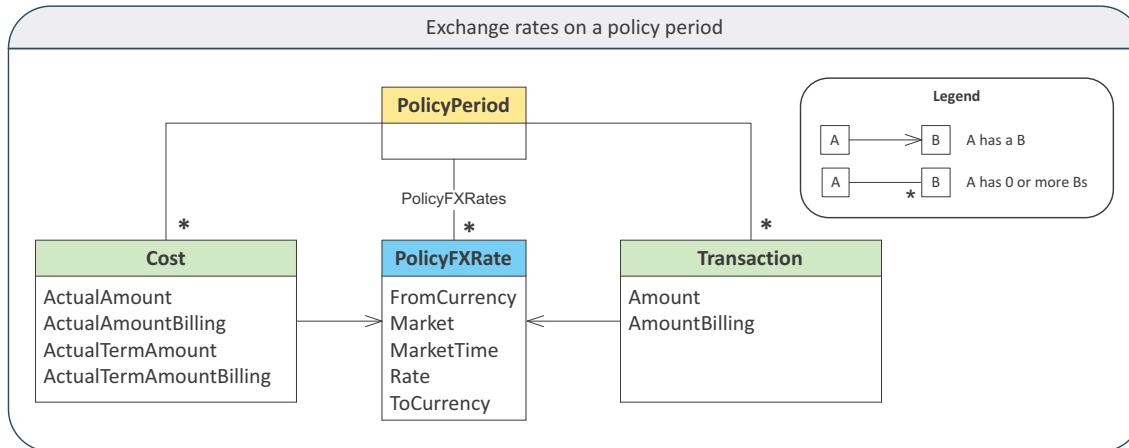
The `convertCostsToSettlementCurrency` method in the `AbstractRatingEngineBase` class performs the currency conversion and stores the converted amounts in duplicate `AmountBilling` properties.

The duplicate amount properties on the Cost entity are:

Property	Description
<code>ActualAmountBilling</code>	The <code>ActualAmount</code> in the settlement currency for accounting and reporting.
<code>StandardAmountBilling</code>	The <code>StandardAmount</code> in the settlement currency for accounting and reporting.
<code>OverrideAmountBilling</code>	The <code>OverrideAmount</code> in the settlement currency for accounting and reporting.
<code>ActualTermAmountBilling</code>	The <code>ActualTermAmount</code> in the settlement currency for accounting and reporting.
<code>StandardTermAmountBilling</code>	The <code>StandardTermAmount</code> in the settlement currency for accounting and reporting.

Property	Description
OverrideTermAmountBilling	The OverrideTermAmount in the settlement currency for accounting and reporting.
PolicyFXRate	If the coverage currency is not the same as the settlement currency, this property contains a foreign key to a copy of the exchange rate from the exchange rate service. PolicyCenter uses this PolicyFXRate to convert the as-rated costs from the coverage currency to billing costs in the settlement currency. When you configure PolicyCenter, do not keep more than one copy of the same PolicyFXRate on a single policy period.

The policy period has an array of exchange rates used for monetary amount conversion. This array is `PolicyPeriod.PolicyFXRates`.



Each FXRate object records an exchange rate and the time that it was obtained from the market. Exchange rates can vary over the life of the policy, come from different sources, or cover different currency pairs, therefore the policy period maintains an array of exchange rates.

The following table describes some of the properties on the `PolicyFXRate` object:

Property	Description
Rate	The exchange rate at which one currency can be converted to another.
MarketTime	The time at which the market indicated the rate went into effect.
RetrievedAt	The time at which the quotation was obtained from the market.
FromCurrency	The currency to convert from.
ToCurrency	The currency to convert to.
Market	The exchange rate market, <code>FXRateMarket</code> , which set the rate.
PolicyPeriod	A foreign key to the policy period to which this object belongs.

## Transactions

In the base configuration, transaction amounts are computed using the as-rated currency amounts on the costs. PolicyCenter then converts the transaction amounts to the settlement currency using the same exchange rate, `PolicyFXRate`, as the cost associated with the transaction. This is one way that a carrier might handle exchange rate changes during a policy term. You can configure PolicyCenter to handle the exchange rate in other ways during transactions.

The `createEntityTransactions` method in the `PolicyPeriodTransactionCalculator` class converts the transaction amount to the settlement currency using the exchange rate. The `PolicyPeriodTransactionCalculator` class is a read-only file.

The properties on the Transaction entity are:

Property	Description
Amount	The transaction amount.
AmountBilling	The Amount in the settlement currency for billing.
PolicyFXRate	A foreign key to a copy of the exchange rate from the exchange rate service. This points to the same exchange rate used to calculate the cost associated with this transaction.

## Configuring Underwriting Authority and Multicurrency

PolicyCenter raises underwriting issues based on characteristics of the policy, including ones that may be related to monetary amounts. You can create and manage monetary underwriting issues with values, approvals, and authority limits that include both an amount and a currency. The base configuration and sample data contain underwriting issues that provide an example of how to compare values, approvals, and authority limits in different currencies.

In the base configuration examples, PolicyCenter creates an issue with a value in the currency on the policy. That currency may differ from the currency of the user's authority grant. In the base configuration, the monetary-issue comparator automatically converts the value being tested to the currency of the reference value. The value being tested might be the value of the user's approval. The reference value currency might be the currency of the user's authority grant. In the sample data, the authority grants are in U.S. dollars, but this is not a requirement. Through configuration, you can take a different approach.

### See also

- “Multicurrency Fields for Underwriting Authority” on page 535 in the *Application Guide*
- “Underwriting Authority” on page 415 in the *Application Guide*
- “Configuring Underwriting Issues” on page 479

## Configuring Underwriting Issues and Multicurrency

In the base configuration, certain underwriting issue types handle multicurrency. An underwriting issue type that handles multicurrency has the following features in the `uw_issue_types.xml` system table:

Field	Description
Comparator	Set to At least or At most. These values correspond to the codes Monetary_GE or Monetary_LE, respectively.
DefaultValueAssignmentType	Set to OffsetAmount.
DefaultValueOffsetAmount	Not equal to 0.
ValueFormatterType	Optional. Set to MonetaryAmount to format the value as an amount and currency in the user interface.

The `PATotalPremium` underwriting issue type is an example that shows multicurrency. In Product Designer, you can view this underwriting issue type in the `uw_issues_types.xml` system table. In the Personal Auto line of business, the evaluator Gosu class (`PA_UnderwriterEvaluator.gs`) determines whether to raise `PATotalPremium` underwriting issues.

As with other underwriting issues, the code that creates the issue determines the value for the issue. PolicyCenter displays this value in the underwriting screens, therefore consider keeping the value (and approval) in the original currency. In the base configuration, the value comparator for monetary issues does a conversion between currencies, if needed. This conversion enables you to specify authority grants in a single currency per issue, yet

have that grant to apply to all currencies for the issue. Then PolicyCenter determines whether or not to raise an underwriting issue. The `UWIssueValueComparatorWrapper` class in the `gw.job.uw` package uses the `MonetaryGEValueComparatorWrapper` and `MonetaryLEValueComparatorWrapper` classes to convert the currency and compare monetary values.

**See also**

- “Configuring Underwriting Authority” on page 469
- “Underwriting Issue Type System Table” on page 489

## Configuring Authority Profiles and Multicurrency

For PolicyCenter users, you set the approval value and currency for these underwriting issue types in an authority grant within an authority profile. Access authority profiles by navigating to **Administration** → **Users & Security** → **Authority Profiles** in PolicyCenter.

**See also**

- “Configuring Authority Grants” on page 478

## Implementing an Exchange Rate Service

Each carrier has their own unique exchange rate requirements such as how often to refresh the exchange rate or which market provides the rates. In some cases, the exchange rate market is not a market in the conventional sense, but a set of rates specified by the company or one of its business associates. A carrier’s exchange rates may vary by line of business. Certain policyholders may have custom rates which are negotiated with the carrier. Therefore, the base implementation of PolicyCenter provides a simple exchange rate service which rates multicurrency policies. In this simple implementation, PolicyCenter obtains the exchange rates from a static table containing rates for conversion from one currency to another. Guidewire expects each carrier to implement their own exchange rate service.

**See also**

- “Exchange Rate for Multicurrency Policies” on page 525 in the *Application Guide*

## The Exchange Rate Service Plugin Interface

The Exchange Rate Service plugin interface (`IFXRatePlugin`) is the interface for obtaining the exchange rate. In the base configuration, the `FXRateServicePlugin` implements this interface. You can use the `FXRateServicePlugin` implementation as a starting point for your own implementation of the `IFXRatePlugin` interface.

The `IFXRatePlugin` interface has the following methods:

Method	Return value	Description
<code>canConvert</code>	<code>Boolean</code>	For two currencies, this method returns true if you can convert from the first currency to the second currency.
<code>getFXRate</code>	<code>FXRate</code>	This method returns the rate for converting from one currency to another. You can optionally specify: <ul style="list-style-type: none"><li>• A date on which the rate was in effect</li><li>• The rate market from which to obtain the rate</li></ul>

## The Built-in Implementation of the Exchange Rate Service Plugin Interface

The built-in implementation of the Exchange Rate Service Plugin (`IFXRatePlugin`) interface is the `FXRateServicePlugin` plugin. This simple implementation gets exchange rates from a static table defined in the plugin. This class is provided only as an example; it is not for production use. Although you can use this plugin as a starting point, you must create your own implementation of the `IFXRatePlugin` interface.

# Enabling Multicurrency Integration

You must perform certain tasks to enable multicurrency integration appropriately.

- “Set up Currency-Specific Plans and Authority Limits” on page 584

## Set up Currency-Specific Plans and Authority Limits

One of the administrative tasks you must perform when setting up BillingCenter is to create plans. Several of these plans are currency specific, and therefore you must set up at least one plan for each currency. There are monetary values that reflect various applicable fees and financial thresholds. For example, billing plans contain an invoice fee. A Euro account must define the fee in euros. As another example, delinquency plans have monetary thresholds that control when BillingCenter makes a policy delinquent. Such thresholds must be defined for each currency.

These currency specific administrative entities that need to be created in BillingCenter are:

- Billing plans
- Payment plans
- Commission plans
- Delinquency plans
- Agency bill plans
- Authority limits

BillingCenter requires all of these be defined to issue a policy in a given currency, except agency bill plans. You must define agency bill plans for each currency in which you plan on issuing policies with agency billing. If you do not define an agency bill plan for a given currency, **Agency Bill** will not appear in PolicyCenter as a choice for billing method.

### See also

- “Plan Types” in the *BillingCenter Application Guide*

## Set up Billing Plans in BillingCenter for Multicurrency Accounts

In BillingCenter, each account must have an associated billing plan. Billing plans determine how BillingCenter handles automatic distributions, special circumstances such as low balance, and invoicing at the account level. Billing plans are currency specific and therefore you must set up at least one billing plan in BillingCenter for each currency.

### See also

- “Working with Billing Plans” in the *BillingCenter Application Guide*

## Set up Payment Plans in BillingCenter for Multicurrency Accounts

In BillingCenter, each policy must have an associated payment plan. Payment plans determine how payments are set up and spread over the term of the policy. Payment plans are currency specific and therefore you must set up at least one billing plan in BillingCenter for each currency. The payment plans that match the currency of a policy appear in PolicyCenter as a list of choices for how to bill a policy.

### See also

- “Working with Payment Plans” in the *BillingCenter Application Guide*

## Set up Agency Bill Plans in BillingCenter for Multicurrency Producer Organizations

In PolicyCenter you set the currency for producer organizations through agency bill plans. PolicyCenter retrieves the agency bill plans from BillingCenter. In a single currency system, you select an **Agency Bill Plan** on the **Basics** tab of the **Organization** screen. In a multicurrency system, you select one or more agency bill plans on the **Agency Bill** tab. Each agency bill plan specifies a currency.

When entering the **Organizations** or the **New Organizations** screens, PolicyCenter synchronizes with BillingCenter to refresh the agency bill plans.

### See also

- “Working with Agency Bill Plans” in the *BillingCenter Application Guide*
- “Organizations and Producer Codes Overview” on page 718 in the *Application Guide*

## Set up Delinquency Plans in BillingCenter for Multicurrency Accounts

A BillingCenter delinquency plan defines a sequence of events to execute when a policy becomes past due. The delinquency plan can be applied to the past-due policy only or to all policies in the account.

A special case exists when all policies in the account are considered delinquent and the account supports multiple currencies. In this situation, the delinquency plan is applied only to the policies that use the same currency as the policy that initiated the delinquency. In other words, only policies contained in the initiating policy’s currency silo are considered to be delinquent. Other policies in the account that use a different currency (and are therefore in a different currency silo) are not considered delinquent. This behavior describes the operation of the BillingCenter base configuration. If you wish all policies in the multicurrency account to be considered delinquent, regardless of their currency, BillingCenter can be configured to support that behavior.

Similarly, when a delinquent account is reinstated, the base configuration reinstates only the policies that reside in the relevant currency silo. To reinstate all policies in the account, regardless of their currency, configuration can be written to support that behavior.

### See also

- “Working with Delinquency Plans” in the *BillingCenter Application Guide*

## Set up Commission Plans in BillingCenter for Multicurrency Producer Codes

In PolicyCenter, producer codes are associated with commission plans. Each commission plan is associated with a currency. PolicyCenter retrieves the commission plans from BillingCenter. In a single currency system, each producer code has a single commission plan. In a multicurrency system, a producer code must have multiple commission plans, one for each currency.

When entering the **Producer Codes** or the **New Producer Code** screens, PolicyCenter synchronizes with BillingCenter to refresh the commission plans for the current producer code.

### See also

- “Working with Commission Plans” in the *BillingCenter Application Guide*

- “Organizations and Producer Codes Overview” on page 718 in the *Application Guide*

### [Set up Authority Limits in BillingCenter for Multicurrency Integration](#)

Each user in BillingCenter needs to be associated to an authority limit profile. BillingCenter supports multicurrency Authority Limit Profiles. Authority Limit Profiles are composed of authority limits, and an authority limit must be defined separately for each currency. For example, one authority limit specifies the maximum amount a user has the authority to write off. If your integration includes Japanese yen, European Union euros, and U.S. dollars, a separate write-off authority limit must be created for each of these currencies. All three of these currency specific authority limits are associated to the same authority limit profile.

#### **See also**

- “Authority Limits and Authority Limit Profiles” in the *BillingCenter Application Guide*

#### **See also**

- “Multicurrency Integration between BillingCenter and PolicyCenter” on page 503 in the *Integration Guide*

# Guidewire PolicyCenter Job Configuration



# Configuring Jobs

Configuring PolicyCenter jobs involves making changes to the product model, Gosu classes and rules, page configuration files (PCF files), wizards, configuration parameters, plugins, and sometimes workflows. Each can be configured based on your business requirements. It is the combination of these elements that allows you to create or modify a policy.

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

In contrast to jobs, the product model defines the types of policies that you can offer to your customers. Each product can contain one or more lines of business offering various coverages. For more information, see “Product Model Overview” on page 475 in the *Application Guide*.

This topic includes:

- “Common Ways to Configure Jobs” on page 589
- “Configuring Specific Aspects of a Job” on page 591

## Common Ways to Configure Jobs

The following topics provide more detail on common ways that you can configure jobs (policy transactions):

- “Wizards for Jobs” on page 590
- “Gosu Classes for Jobs” on page 590
- “Rule Sets” on page 591
- “Workflows” on page 591
- “System Configuration Parameters for Jobs” on page 591

## Wizards for Jobs

A wizard is a type of PCF file that contains a series of user interface screens. Wizards collect information and perform operations while guiding the user through a complex business transaction.

Each job (policy transaction) has a wizard associated with it. For example, the submission job has the `SubmissionWizard.pcf` file which contains the steps for collecting information about a submission. The wizard pre-qualifies the applicant, collects policy information, performs risk analysis, reviews the policy, views a quote, prints forms, and sets up billing.

Each wizard step has a screen such as the **New Submissions**, **Pre-Qualification**, **Policy Info**, **Quote**, and **Submission Bound** screens. Each screen is a PCF file that defines the screen and its component parts.

As the user moves through the wizard, the wizard calls functions in the job code for processing and may indirectly access rule sets and workflows. The job code contains most of the code for the job and also interacts with the rule sets and workflows. In some cases, the wizard accesses the workflow directly.

You can modify wizards to gather the data that you want users to collect while processing a job. In Studio, you can access the PCF and wizard files for jobs by navigating to **Page Configuration** → **pcf** → **job**.

### See also

- “Introduction to Page Configuration” on page 303

## Gosu Classes for Jobs

The Gosu job code contains most of the programming logic for the job (policy transaction). In Studio, you can view the code for each job type by navigating to **Configuration** → **gsrc** → **gw** → **job**. The types of files that make up the job code are:

- **Job Class** – Each job type defines a subclass of the `JobProcess` class. For example, the `SubmissionProcess.gs` file defines the `SubmissionProcess` class. This class defines the steps of a job. It also defines the core functionality and core properties of the job, such as request quote or bind. The job class is marked as `@Export`, so you can modify the class directly.

Be careful when you modify the job class because it is the primary controller for how jobs work. Modifications can easily cause the job process to break.

For information about classes, see “Classes” on page 191 in the *Gosu Reference Guide*.

- **Job Subclass** – Rather than modifying the job class directly, you can indirectly change or add methods and properties to the job by defining a subclass. You may find it easier to maintain changes in a subclass rather than the job class because the subclass contains your code only.

For example, to create a subclass for the submission class, you create a file such as `YourCompanySubmissionProcess.gs` that extends the `SubmissionProcess` class:

```
class YourCompanySubmissionProcess extends SubmissionProcess {
 ...
}
```

Be careful when you modify a subclass because it modifies the job process and can easily cause the job process to break.

If you create a subclass, you must modify the job process customization plugin. For more information, see “Job Process Creation Plugin” on page 152 in the *Integration Guide*.

- **Enhancement** – You can use a Gosu enhancement to add methods and properties to a class. These methods and properties are available to all objects of the enhanced type. The default application contains some enhancements to the job entities, and you may modify these or add your own enhancements. Enhancements often contain code that is not the core functionality of the job and that you can customize to meet your business needs.

If you need to modify the enhancement for the submission class, edit `SubmissionEnhancement.gsx`:

```
enhancement SubmissionEnhancement : Submission {
 ...
}
```

**IMPORTANT** The job class code may change in future releases. Therefore, be sure to use comments in the code to clearly delimit and document your changes. As part of moving to a new release, you may have to implement (not just upgrade) your changes again. Review your modifications to the job class and enhancement as well your modifications to job subclasses.

#### See also

- “Enhancements” on page 229 in the *Gosu Reference Guide*

## Rule Sets

Another way to add business logic for jobs and workflows is through rule sets. You can use rules to check for a simple set of conditions and then follow with a set of simple actions. Rule sets are grouped by function for the purpose of customizing a process, such as a submission, to follow established business procedures.

**Note:** Instead of using rules, you can also add your business logic in the Gosu job process code. Inserting logic in the job code allows for a more centralized job flow and is easier to debug. The default application puts most of the business logic in the Gosu job process code.

#### See also

- “Introduction to Business Rules” on page 11 in the *Rules Guide*

## Workflows

PolicyCenter uses workflows for asynchronous steps and for automated processes. In the default installation, only renewal and cancellation jobs have workflows. Cancellation has a workflow that handles the potential waiting period between the time the cancellation is scheduled or issued and completed. Renewal has workflows to handle automated renewals.

The workflow calls and is called by the job code and by rules in the rule sets. In some cases, the wizard calls the workflow and sends control back to the wizard.

#### See also

- “Guidewire Workflow” on page 373

## System Configuration Parameters for Jobs

There are configuration parameters that determine the behavior of the various jobs. You can find these parameters in `config.xml` which you can view in Studio by navigating to `configuration → config`. For more information about these parameters see:

- “Miscellaneous Job-Related Parameters” on page 64
- “Job Expiration Parameters” on page 62

## Configuring Specific Aspects of a Job

The following topic describes how to configure specific aspects of a job such as:

- “Changing Jobs to Expired Status” on page 592
- “Configuring Job History Events” on page 592

- “Multiple Revision Jobs and the Job Selected Branch Property” on page 595
- “Selecting the Underwriting Company through Segmentation” on page 595

## Changing Jobs to Expired Status

A job in an **Expired** status indicates that it has been closed due to inactivity. An expired job is in a terminal state, and you cannot be reopen it. However, you can clone an expired job into a new, open job.

The **Job Expire** batch process changes jobs from **New**, **Draft**, or **Quote** status to **Expired** if they have passed an expiration threshold. In the default configuration, the **Job Expire** batch process expires submission jobs with these statuses that are at least seven days past the effective date of the policy. In addition to submission jobs, you can enable expiration for other job types. You can configure the expiration threshold as the number of days past the effective date or the creation date.

Expiring jobs at an appropriate time can impact searching and reporting, and have a critical impact on business operations including purging and archiving. Select an expiration threshold that is consistent with the intended behavior of impacted business operations.

A job is expired after a sufficient and configurable interval of time has elapsed. A policy version can be expired when its status is either **New**, **Draft**, or **Quoted**. If the policy version is already closed by being withdrawn, issued, or not taken, then it will not be expired because it is already closed. When a policy version expires, its status changes to **Expired**. You can view the status in the **Submission Manager** screen or in the toolbar if you are in the submission wizard. A branch is not editable.

### Configuring the Batch Process

The **Job Expire** batch process triggers job expiration. The batch process, by default, runs every day at 6 A.M. as configured in the `scheduler-config.xml` file. The batch process expires all versions of a policy in a job that can be expired.

You can configure both the scheduling and the amount of time that must elapse before expiration. Configure the batch process by modifying the following parameters in the `config.xml` file:

- `JobExpirationEffDateThreshold` is the number of days past the effective date of the policy version before a job can be expired. PolicyCenter compares this value with the effective date of the policy version, the default value is 7 (a week).
- `JobExpirationCreateDateThreshold` is the number of days past the create date of the submission before a submission can be expired. The create date is the day the submission entered the system. PolicyCenter compares this value with `Job.CreateTime`.

#### See also

- “Scheduling Work Queues and Batch Processes” on page 107 in the *System Administration Guide* for more information on how to configure the **Job Expire** batch process.
- “Job Expiration Parameters” on page 62

## Configuring Job History Events

History events record the progress of a job (policy transaction). PolicyCenter displays history events in the **History** menu link in the **Policy** tab. In the default configuration, the submission and renewal jobs log history events. In addition, creating a new account and changing an account creates history events. You can add history events to a job by using the `createCustomHistoryEvent` method of the `Job` class. This method has two signatures:

- The first `createCustomHistoryEvent` method creates a `History` with the given `CustomHistoryType` and `description`. The `description` argument is wrapped in a block so that it can be localized to the primary language of the policy or account.

```
function createCustomHistoryEvent(type : CustomHistoryType, description : block() : String) : History
```

- The second `createCustomHistoryEvent` method is similar, but allows you to provide the `originalValue` and `newValue` fields of the History.

```
function createCustomHistoryEvent(type : CustomHistoryType, description : block() : String,
 originalValue : String, newValue : String) : History
```

For examples, view the job classes, such as `RenewalProcess.gs`, in Studio.

The History screen displays a **Type** and **Description** for each event. In Studio, type names are defined in the `CustomHistoryType.ttx` typelist. Descriptions are defined in **Display Keys**. The submission job display keys are defined in `Submission.History` entries. The renewal job display keys are defined in `Job.Renewal.History` entries. You can localize both the type and description.

### Example Code that Localizes the Event Message to the Primary Language

The `description` argument of `createCustomHistoryEvent` is wrapped in a block so that it can be localized to the primary language of the policy or account. This example describes the code that does this.

Code in `CancellationProcess.gs` calls the `Job.createCustomHistoryEvent` method:

```
private function startScheduledCancellation(processDate : Date) {
 ...
 if (InitialNotificationsHaveBeenSent) {
 Job.createCustomHistoryEvent(CustomHistoryType.TC_CANCEL_RESCHEDULE, \ ->
 displaykey.Job.Cancellation.History.Reschedule(processDate))
 } ...
}
```

The second parameter of `createCustomHistoryEvent` method is defined as:

```
description : block() : String
```

The `description` parameter is a method, `block()`, that returns a `String`. In code above, the parameter is the `displaykey.Job.Cancellation.History.Reschedule` method.

The `createCustomHistoryEvent` method contains code which temporarily switches the locale to the primary language of the policy and runs the `description` method. Therefore, the history event message is localized to the language of the policy.

```
LocaleUtil.runAsCurrentLocale(LocaleUtil.toLocale(this.Policy.PrimaryLanguage), \ -> {
 history.Description = description()
}
```

#### See also

- “Gosu Blocks” on page 233 in the *Gosu Reference Guide*
- “Localizing Display Keys” on page 45 in the *Globalization Guide*

### History Events in the Default Configuration

In the default configuration, history events are logged when changes are made in an account, job, and policy term.

#### Account

PolicyCenter logs the following history events in an account:

- An **Account created** event is created when new accounts are created through the user interface or the `AccountAPI`.
- An **Account changed** event is created:
  - When the account status changes from **Pending** to **Active** at the start of a submission.
  - When the account holder is changed to a different contact. The event contains the old and new values.

You can add additional history events by configuring PolicyCenter.

#### Job

PolicyCenter logs the following history events for jobs (policy transaction):

- Cancellations
  - A **Cancel reschedule** event is created when a cancellation is rescheduled.
- Policy change
  - A **PolicyChange created** event is created when the job is created.
  - A **PolicyChange Effective Date change** event is created when the edit effective date of the policy change changes.
- Questions
  - An **Answer changed** event is generated when you change an incorrect answer to a blocking question to a correct answer.
- Renewals
  - A **Renewal** event is created for numerous events in a renewal job. Since renewals are typically automated, the histories provide insight into the progress of the job. The **Renewal** event is created:
    - When an automated renewal is escalated or manually edited.
    - When a user presses the **Renew**, **NonRenew**, **Not taken**, or **Issue Now** buttons.
    - When the automated renewal enters the pending renew, pending non-renew or pending not taken states.
    - If the product is no longer available and the renewal will be non-renewed.
    - When the renewal is quoted.
    - When the renewal is withdrawn.
    - When the renewal completes issuance, non-renewal, or not taken.
- Rewrite
  - A **Rewrite created** event is created when the rewrite job starts.
- Submissions
  - A **Submission bound** event is generated when the job is bound.
  - A **Submission created** event is generated when the job is created.
  - A **Submission decline** event is generated when the job is declined.
  - A **Submission issued** event is generated when the job is issued.
  - A **Submission quoted** event is generated when the job is quoted.

### Policy Term

PolicyCenter logs the following history event for policy terms:

- Pre-renewal
  - A **Pre-renewal** event is created when a user sets the pre-renewal direction of a policy.
  - A **Pre-renewal** event is created when a user non-renews a renewal. Although the non-renewal action occurs in the renewal job, the history event is added to the **PolicyTerm**. The history description in the user interface is made by concatenating certain pre-defined strings when the user adds or removes a non-renewal explanation pattern.

### Configuring History Search Criteria

You can configure the search criteria for history events in the Gosu class `gw.history.HistorySearchCriteria`.

#### See also

- History in “Policy Tools Menu” on page 309 in the *Application Guide*
- “Account File History Screen” on page 333 in the *Application Guide*

## Multiple Revision Jobs and the Job Selected Branch Property

To support multi-revision jobs (policy transactions), there is a property `Job.SelectedVersion`, which indicates for multi-revision jobs which branch is the selected branch. Knowing which branch is the selected branch is important especially where there is more than one non-withdrawn branch in the job.

There is always a selected branch for any job. The selected branch is always one of the non-withdrawn branches, if one or more remain. This makes the selected branch a safe way to obtain a `PolicyPeriod` from a `Job`. The selected branch may or may not be the one that the user is actively modifying in the user interface. The selected branch depends upon the user's actions and any business logic that has been implemented. In some display and reporting logic, the selected branch is the one that is used when more than one branch exists.

The `SelectedVersion` property is an edge foreign key from `Job` to `PolicyPeriod`. In the database, this creates a join table called `pc_jobpolicyperiod` where the `OwnerID` column points to the job, and the `ForeignEntityID` property points to the `PolicyPeriod`.

If you withdraw a revision from a multi-revision branch and there is only one non-withdrawn branch, PolicyCenter automatically sets that branch to the active branch.

In the user interface for multi-revision jobs, you can select one revision by clicking **Selected**.

## Selecting the Underwriting Company through Segmentation

Guidewire PolicyCenter provides the ability to set the underwriting company for a policy on the `Policy Info` page for Issuance, Rewrite, Renewal, and Submission jobs.

- If the user has the requisite underwriting permission (Multiple company quote), then PolicyCenter displays a list of valid underwriting companies. The user can then select from the list and set the underwriting company.
- If the user does not have the necessary underwriting permission, then PolicyCenter displays an **Autoselect** button instead.

### Segmentation Example

Segmentation classes determine the segmentation, and, therefore, the available underwriting companies for a policy. Large carriers typically license more than one subsidiary to underwrite policies on behalf of the carrier. The primary reason for having multiple subsidiaries is to accommodate state regulatory requirements. (Many states do not allow carriers to have more than one set of rates per underwriting company.) Therefore, if a carrier desires to offer multiple sets of rates in that state, the carrier must file each set of rates under a separate underwriting company for that state.

Underwriters typically profile an account to determine the segment (category) into which an account falls. The segment determines which underwriting company actually underwrites the policy. By extension, segmentation sets the rates for which an account is eligible. The underwriter then uses appropriate set of rates to quote the policy.

For example, a workers' compensation carrier can segment accounts into three different segments:

- High hazard
- Medium hazard
- Low hazard

The carrier has three underwriting companies corresponding to each of the segments. To generate a quote for the policy, an underwriter indicates the underwriting company with which to place the policy. The rating engine takes this information and uses it to calculate a quote for the policy with the appropriate set of rates.

In the following example from `WC_SegmentEvaluator`, segmentation sets the segmentation level for any policy written for the state of Hawaii to high risk.

```
override property get IsHighRisk() : boolean {
 return _policyPeriod.PrimaryLocation.State == "HI"
}
```

## Segmentation Classes

The following Gosu classes implement segmentation:

- `SegmentEvaluator` – The interface
- `AbstractSegmentEvaluator` – Implements the `SegmentEvaluator` interface
- Subclasses – `DefaultSegmentEvaluator` and line specific ones like `WC_SegmentEvaluator` that is created from `PolicyLineMethods#createSegmentEvaluator`.

## PolicyPeriodBaseEnhancement

**Note:** Guidewire intends that you customize this enhancement to meet your specific business needs.

The `PolicyPeriodBaseEnhancement.gsx` enhancement (`gw.policy.PolicyPeriodBaseEnhancement`) contains code for auto-selecting an underwriting company. In particular, it contains the following method:

```
autoSelectUWCompany
```

This method contains simple logic for determining the best company to be set as the underwriting company for the `PolicyPeriod` based on segment and lowest price.

A user triggers the `autoSelectUWCompany` method within PolicyCenter if the user clicks **AutoSelect** for underwriter. (As mentioned, this button is available only if the user does not have the `Multiple company quote` underwriting permission.)

You set this method on `PolicyPeriod` in Studio, in any of the following PCF files:

- `IssuanceWizard_PolicyInfoDV`
- `RenewalWizard_PolicyInfoDV`
- `RewriteWizard_PolicyInfoDV`
- `SubmissionWizard_PolicyInfoDV`

To see this in the base configuration, open one of these files (`IssuanceWizard_PolicyInfoDV`, for example). Then, find the **Autoselect** button on the PCF page and select it. View the action property for this widget (at the bottom of the page). You see the following:

```
policyPeriod.autoSelectUWCompany()
```

## getUWCompaniesForStates

Use the following method to return a set of underwriting companies based on the effective dates and all the states needed for the `PolicyPeriod`.

```
PolicyPeriod.getUWCompaniesForStates(boolean allStates)
```

For the `allStates` parameter:

- If `true`, the method returns only the underwriting companies that are valid for all states for the `PolicyPeriod`. For example, suppose that the company requesting the policy does business in three different states (Colorado, New Mexico, and Arizona). You might, therefore, want to find only those underwriting companies that can do business in all three of those states.
- If `false`, the method returns underwriting companies that are valid for any state for the `PolicyPeriod`. For example, suppose that you have the same company that does business in Colorado, New Mexico, and Arizona. This time, you only want to find the set of underwriting companies that do business in at least one of those states. You might want this, for example, if you wanted to split the policy between multiple underwriting companies based on the state.

To verify that you have a valid underwriting company, use the following method:

```
PolicyPeriod.getUWCompaniesForStates(true).contains(myUWCompany)
```

# Configuring Submissions

This topic describes how to configure submission jobs (submission policy transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

- “Configuring Submissions Overview” on page 597
- “Submission Integration” on page 599
- “Submission Configuration Examples” on page 601

**See also**

- “Submission Policy Transaction” on page 79 in the *Application Guide*
- “Configuring Jobs” on page 589
- “Selecting the Underwriting Company through Segmentation” on page 595

## Configuring Submissions Overview

Use Studio to view and edit the Gosu classes and other files related to the renewal process.

Key files include:

- **Gosu classes** such as: `SubmissionProcess.gs`. This class defines and begins the new submission process. You can view and edit this class by navigating in Studio to **Classes** → **gw** → **job**, and finding the appropriate class.
- **Gosu enhancement files** such as: `SubmissionEnhancement.gsx`. This enhancement allows you to augment classes and other types with additional methods and properties. An example is the `addToGroup` method which groups it into the appropriate Submission group or creates a new one if a valid group does not exist.
- **Page configuration files (PCF files)** such as `SubmissionWizard.pcf` or `SubmissionWizard_PaymentScreen.pcf`.

- **TypeLists** such as JobGroup, or QuoteType.

**Note:** Since the submission process for any carrier can be configured based on business requirements, all discussions apply to the default application only.

This topic includes:

- “Submission Process Gosu Class” on page 598
- “Submission Enhancements” on page 599

## Submission Process Gosu Class

Gosu classes contain the code which primarily controls the submission process. In Studio, navigate to `configuration → gsrc` and then open the `gw.jobSubmissionProcess.gs` class. This is the primary class for submissions. The job classes are part of the default application.

The `SubmissionProcess.gs` class contains the main actions involved in a submission and extends the `JobProcess` Gosu class. The `JobProcess` class contains methods that allow you to get the job, the policy period, and a quote if it exists. It cancels any open activities on the submission, auto assigns the user role, and checks to see if the job can be expired, withdrawn, or approved by an underwriter. It also checks to see if the job can be started as new.

The `SubmissionProcess.gs` starts a new submission and assigns a requestor and producer. The `beginEditing` method defines and enacts the transaction for a new submission that moves it to `Draft` status. Segmentation classes determines the available underwriting companies for a policy.

Next, the submission process determines whether quoting can be initiated. The process determines if the branch can be quoted in its current state as seen through the `canRequestQuote` and `canEnterQuoting` methods respectively. It also determines whether the submission needs to be reviewed by an underwriter. If the submission began as a quick quote, it also converts it to a full application. That in turn allows a job to be bound `bindOnly`, `bindAndHoldIssuance`, and `issue`.

After PolicyCenter starts the bind process for the submission (branch), the submission may be rejected, marked for review by an underwriter, or returned to `Draft` for additional editing. Otherwise, binding begins, which puts the submission in the `Binding` status, updates the policy number, and sends an event for additional processing.

To complete binding, PolicyCenter calls the `finishBinding` method. This method completes the binding of a policy period without issuing it. The policy period is marked as bound and the job is finished. However, if issuance is being held, then PolicyCenter promotes the branch to the main policy period. Branches with no hold status must go through the `finishIssuing` method. This method promotes the policy period by marking it `Bound` and the completes job.

After issuing a submission, the user can bill the submission. The billing system plugin interface `IBillingSystemPlugin` defines the contract between PolicyCenter and the billing system. For example, the billing plugin implementation sends messages to the billing system about new billing instructions associated with a new submission.

**Note:** The submission job no workflows because it has no background processes. PolicyCenter uses workflows only for background processes.

## Modifying the Submission Process

If you need to change or add methods to the submission process, you can edit the job class directly or create a subclass (such as `YourCompanySubmissionProcess.gs`).

For more information about job classes and job subclasses, see “Gosu Classes for Jobs” on page 590.

If you create a subclass, you must modify the job process customization plugin. For more information, see “Job Process Creation Plugin” on page 152 in the *Integration Guide*.

## Submission Enhancements

There are other submission enhancements that contain additional functionality. You can modify these files.

- `SubmissionEnhancement.gsx` – This enhancement creates a copy of a submission and commits it. However, PolicyCenter does not validate the copied submission during the commit. In the `copySubmission` method, you can insert code for any additional copying. For fields that you do not want to copy, you can erase or reset them to default values. The `addToGroup` method groups a submission job into the appropriate submission group or creates a new one if a valid group does not exist.
- `SubmissionGroupLettersEnhancement.gsx` – This enhancement returns an array of the producers of the submissions in the submission group. It also verifies, through the `canAnySubmissionSendLetter` method, if any submission in the batch can have a letter generated for it. If it does, then you see that option in the user interface. Lastly, there is a `sendConfirmationLetter` that is a placeholder for implementing through integration, a confirmation letter. Its purpose is to send a confirmation letter with every possible submission in the batch.

For more information about job enhancements, see “Gosu Classes for Jobs” on page 590.

## Submission Integration

PolicyCenter integrates with many types of external systems through a diverse toolbox of services that can link PolicyCenter with custom code and external systems. This topic describes web services and plugins that the submission job uses.

### Submission Web Services

`JobAPI` primarily adds an activity to a job by using an activity pattern. It also has methods to find and withdraw jobs.

`PolicyPeriodAPI` is a web service for performing various operations on policy periods within PolicyCenter. There are methods to add the following:

- Add a note to the policy and policy period
- Add a document to the policy period
- Add a referral reason code to the policy period

`WorkflowAPI` is a web service for performing various operations on workflows within PolicyCenter. It can:

- Start an action (by invoking a trigger) to advance the workflow to the next step.
- Resume a workflow or all the workflows. The workflow engine can attempt to advance the workflow (or all workflows) to the next step. If there is an error, then it is logged.
- Suspend a workflow.

### Submission Plugins

Submissions use the following plugins.

Name	Description
<code>IAccountPlugin</code>	This account plugin interface contains methods for detecting existing accounts, performing name clearance, generating account numbers, and doing risk reservation.
<code>IBillingSystemPlugin</code>	This plugin notifies an external system about billing events.

Name	Description
IEffectiveTimePlugin	<p>This plugin determines the initial time component of effective/expiration dates (the PeriodStart/PeriodEnd dates) for a policy period when starting a new job. Submission is one of the affected jobs. All of the methods of this interface have a <code>PolicyPeriod</code> parameter, which can be used to access additional information that might be used to determine the desired time, such as:</p> <ul style="list-style-type: none"> <li>• The line or lines of business</li> <li>• The base jurisdiction</li> <li>• Other revisions</li> </ul> <p>The return value from each method is a <code>Date</code>, with its time component, <code>HH:mm:ss</code>, set to the desired time. The plugin discards the day component, <code>dd/MM/yyyy</code>. As a convenience, each method returns null to leave the time component set to the default (midnight).</p>
IJobNumberGenPlugin	This plugin generates a new unique job number.
AccountLocationPlugin	For account locations and policy locations, customize how PolicyCenter: <ul style="list-style-type: none"> <li>• Determines that two locations are the same</li> <li>• Clones a location</li> </ul>
IPolicyNumGenPlugin	This plugin returns an identifier for a new period of a policy. The policy number may vary from period to period. Call this plugin for the initial period of a policy as part of a Submission job. You can also call this plugin if you want to generate a new identifier when renewing or rewriting a policy.
IPolicyPeriodDiffPlugin	<p>This plugin does the following:</p> <ul style="list-style-type: none"> <li>• Compares two policy periods, returning a list of <code>DiffItems</code> representing the differences between them. The following call the <code>compareBranches</code> method:           <ul style="list-style-type: none"> <li>• Multi-version quote</li> <li>• Comparing <code>PolicyPeriods</code> of a Policy</li> <li>• Comparing Jobs of a Policy</li> <li>• Filters a list of <code>DiffItems</code> that originate from the database.</li> </ul> </li> </ul>
IPolicyPeriodPlugin	This plugin mainly calculates period end dates and creates the job process for that period.
ITerritoryCodePlugin	This plugin returns the territory associated with the passed-in search criteria. The territory contains a code and a physical place. The plugin requires that the criteria has non-null values for jurisdiction, policy line pattern, and effective date.
IVinPlugin	For a VIN (Vehicle Identification Number), the <code>getVehicleInfo</code> method returns the model, make, year, and color of vehicle.

### See also

- “Overview of PolicyCenter Plugins” on page 124 in the *Integration Guide* for information about these plugins

## Expiring Submissions

The Job Expire batch process changes submission jobs from `New`, `Draft`, or `Quote` status to `Expired` if they have passed an expiration threshold. In the base configuration, the batch process expires submission jobs with these statuses that are at least seven days past the effective date of the policy. The batch process, by default, runs every day at 6 A.M. as configured in the `scheduler-config.xml` file. The batch process expires all versions of a policy in a job that can be expired. If the policy version is already closed by being withdrawn, issued, or not taken, then it will not be expired.

### See also

- “Changing Jobs to Expired Status” on page 592

## Submission Configuration Examples

The following examples show the configuration of features from the base application, and provide guidance on how to configure the features to meet your requirements.

### Configuring the Copy Submission Feature

You can copy any single PolicyPeriod on a submission. To modify the code, open Studio, and navigate to the classes → jobs. Open the `SubmissionEnhancement.gsx` file and find the method `copyPolicyPeriod`.



# Configuring Issuance

This topic describes how to configure issuance jobs (issuance policy transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

- “Configuring Issuance Overview” on page 603
- “Issuance Integration” on page 604

**See also**

- “Selecting the Underwriting Company through Segmentation” on page 595

## Configuring Issuance Overview

The Gosu class, `gw.job.IssuanceProcess.gs`, contains the main logic of an issuance job. In Guidewire Studio, navigate to **configuration** → **Classes** to view it.

The `hasEditPermission`, `hasQuotePermission`, and `hasWithdrawPermission` methods check to see if you can make any edits, requote the policy, or if necessary, withdraw it.

The `beginEditing` method starts the process of issuing by moving the policy to Draft status so it can be changed if necessary. Like the submission job, it also invokes segmentation.

The `issue` method starts issuance for the policy period and withdraws any other active periods. The policy has an Issuing status. The method also sends the policy to the `FormInferenceEngine.gs` class. The `FormInferenceEngine` serves as the main access point for the rest of the application logic into the form inference logic.

**Note:** The issuance job no workflows because it has no background processes. PolicyCenter uses workflows only for background processes.

## Modifying the Issuance Process

If you need to change or create new methods in `IssuanceProcess` class, you can edit the class directly or create a subclass (such as `YourCompanyIssuanceProcess.g`s).

For more information about job classes and job subclasses, see “Gosu Classes for Jobs” on page 590.

If you create a subclass, you must modify the job process customization plugin. For more information, see “Job Process Creation Plugin” on page 152 in the *Integration Guide*.

## Issuance Integration

Issuance integrates with many types of external systems through a diverse toolbox of services and APIs that can link PolicyCenter with custom code and external systems. This topic describes APIs, plugins, and events that the issuance job uses. There are multiple integration points that an issuance job requires. Most likely, you will integrate to a print issuance system.

### Issuance Web Services and Plugins

The issuance job has the following web services and plugins.

#### Web Services

There are no web services for issuance. The `JobAPI` and `PolicyPeriodAPI` apply to jobs in general.

#### Plugins

Issuance uses the following plugins.

##### Billing System Plugin

When you complete an issuance policy transaction in the PolicyCenter user interface, PolicyCenter calls the `finishIssuing` method, and the method performs some final checks. The method changes the policy status to `Bound` and invokes the `IBillingSystemPlugin`.

The billing system plugin interface, `IBillingSystemPlugin`, defines the contract between PolicyCenter and the billing system. This plugin is the primary mechanism for PolicyCenter to get information from a billing system or to notify the billing system of changes to a policy.

# Configuring Renewals

This topic describes how to configure renewal jobs (renewal policy transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

- “Configuring Renewals Overview” on page 605
- “Renewal Integration” on page 608
- “Configuring Batch Process Renewals” on page 610
- “Configuring Explanations in Pre-renewal Directions” on page 613

**See also**

- “Renewal Policy Transaction” on page 93 in the *Application Guide*
- “Selecting the Underwriting Company through Segmentation” on page 595

## Configuring Renewals Overview

In PolicyCenter, you can configure the renewal job (policy transaction) through Studio. The job process is controlled by:

- **Gosu classes** such as: `RenewalProcess.gsx`. This class contains the main functionality of the renewal process.
- **Gosu enhancement files** such as: `RenewalEnhancement.gsx`. This enhancement allows you to augment classes and other types with additional methods and properties. An example is the `addToGroup` method which groups the job into the appropriate renewal group or creates a new one if a valid group does not exist.
- **Workflows** such as `StartRenewalWF`.
- **Page configuration files (PCF files)** such as `PreRenewalDirectionPage.pcf`.
- **Typelists** such as `RenewalCode`.

- **Rule Sets** such as Renewal AutoUpdate.
- **System tables** such as the NotificationConfig system table.

**Note:** Since the renewal process for any carrier can be configured based on business requirements, all discussions apply to the default application only.

## Renewal Process Gosu Class and Enhancements

Use Studio to view and edit the Gosu classes related to the renewal process.

### Renewal Process Class

The `RenewalProcess.gs` class is the main class controlling the renewal process. In Studio, you can view or edit the class by navigating to `configuration → gsrc`, and opening `gw.job.RenewalProcess.gs`.

This class contains the main logic of the renewal job, including referral direction handling. PolicyCenter checks multiple methods to see if the renewal can be automatically processed based on certain conditions. If PolicyCenter cannot process the job automatically, it blocks the renewal from advancing, raises an issue, and refers the renewal to an underwriter so that it can be manually processed.

**Note:** The `JobProcess.gs` Gosu class contains methods that are common to all jobs, including renewal.

You can view that class in Studio and see the complete list of methods.

The renewal process class includes methods that do the following:

- Starts the renewal
- Sends renewal documents
- Escalates the renewal
- Issues and binds the renewal
- Handles user actions such as:
  - **Issue Now** – Immediately binds the policy.
  - **NonRenew** – Puts the policy period in NonRenewing status.
  - **Not Taken** – Puts the policy period in NotTaking status.
  - **Renew** – Puts the policy period in Renewing status.
  - **Edit Policy Transaction** – Puts the period back in Draft mode for editing and checks the conditions for which a new version of the policy period can be created.
  - **Withdraw Transaction** – Withdraws the policy period. If the renewal is at a point where it cannot be withdrawn, then a message appears stating that the renewal cannot be withdrawn.

### See also

- See “Classes” on page 191 in the *Gosu Reference Guide* to learn how to work with Gosu.
- See “Interfaces” on page 213 in the *Gosu Reference Guide* to learn how to work with interfaces.

### Modifying the Renewal Process Class

If you need to change or create new the methods in that class, you can edit the class directly or create a subclass (such as `YourCompanyRenewalProcess.gs`).

For more information about job classes and job subclasses, see “Gosu Classes for Jobs” on page 590.

If you create a subclass, you must modify the job process customization plugin. For more information, see “Job Process Creation Plugin” on page 152 in the *Integration Guide*.

## Renewal Enhancement

Enhancement files are editable, and you can add additional logic to the renewal process in `RenewalEnhancement.gs`.

## Renewal Workflows

**Note:** PolicyCenter uses workflows in certain job (policy transaction) situations. It is not necessary to use workflows to start a job. Only job steps that are asynchronous use workflows. For synchronous steps, Gosu classes and enhancements contain the logic.

In the default configuration of PolicyCenter, if the renewal process job is automated and not dependent on user actions, then a workflow controls the process. However, if manual intervention is necessary, then the renewal workflow is not used. The renewal process class handles manual renewal steps. See “Renewal Process Class” on page 606.

The renewal job uses the following workflows:

- “Start Renewal Workflow” on page 607
- “Renewal Timeout Workflow” on page 607

### Start Renewal Workflow

In the `RenewalProcess.gs` class, the `start` method begins the `StartRenewalWF` workflow.

This simple workflow has `Begin` and `Done` steps. The `Begin` step gets the user which the automated renewal will use, and starts the automatic renewal by calling the `beginAutomaticRenewal` method defined in `RenewalProcess.gs`.

### Renewal Timeout Workflow

The `scheduleTimeoutOperation` method calls the Renewal Timeout workflow. Various methods in the renewal process class call this method to:

- Wait until a timer expires
- Edit the Policy and change the policy to draft mode.
- Withdraw the policy
- Issue the policy renewal

#### See also

- “Renewal Process Class” on page 606

## Renewal Rule Sets

In PolicyCenter, you can also embed your business logic in rule sets. Use the rule sets if there is a need to check for a simple set of conditions. Follow with a set of simple actions if those conditions are met. Rule sets are grouped by function for the purpose of customizing a process, such as a renewal, to follow established business procedures. Edit rules in Studio in the `configuration → config → Rule Sets` directory.

**Note:** You can also add your business logic in Gosu, for example, putting additional renewal business logic in the `RenewalEnhancement.gs` enhancement. Inserting logic in Gosu allows for a more centralized job flow and is easier to debug.

See the Renewal topic in “*Rule Set Categories*” on page 33 in the *Rules Guide* for additional information.

The following table lists the rule sets that PolicyCenter provides in the base configuration.

Rule set	Rule	Acts on	Description
Renewal	RenewalAutoUpdate	PolicyPeriod	Called when PolicyCenter creates a renewal job. This rule set can be used to perform automatic changes to policy information that need to occur during a renewal.

## Renewal Integration

PolicyCenter integrates with many types of external systems through a diverse set of services. This topic describes web services, plugins, and events that apply to the renewal job (policy transaction).

### Renewal Web Services and Plugins

This topic describes the web services and plugins for renewal.

#### Web Services

Renewal jobs have the following APIs.

Name	Description
PolicyRenewalAPI	<p>The policy renewal web service provides methods for managing renewals in PolicyCenter.</p> <p>The policy renewal web service provides methods to:</p> <ul style="list-style-type: none"> <li>• Start renewals on policies that exist in PolicyCenter. There is no gap in coverage between the renewal and its based-on policy period.</li> <li>• Import a policy from a legacy system and renew the policy. There is no gap in coverage between the renewal and its based-on policy period.</li> <li>• Process a renewal in a billing system</li> </ul> <p>For more information see “Policy Renewal Web Services” on page 115 in the <i>Integration Guide</i>.</p>
JobAPI	<p>The job web service primarily adds an activity to a job by using an activity pattern. This web service also has methods to find and withdraw jobs.</p> <p>The job ID of the activity is set to the given job ID. The previous UserID of the activity is set to the current user. The Assignment Engine assigns the newly created activity to the specified group and user. Finally, the activity is saved to the database, and the public ID of the newly created activity is returned.</p> <p>For more information, see “Job Web Services” on page 104 in the <i>Integration Guide</i>.</p>
PolicyPeriodAPI	<p>The policy period web service provides methods for performing various operations on policy periods within PolicyCenter. It allows you to do the following:</p> <ul style="list-style-type: none"> <li>• Add a note to the policy and policy period</li> <li>• Add a document to the policy period</li> <li>• Add a referral reason code to the policy period</li> </ul> <p>For more information, see “Policy Period Web Services” on page 112 in the <i>Integration Guide</i>.</p>
WorkflowAPI	<p>The workflow web service provides methods for performing various operations on workflows within PolicyCenter. This web service can:</p> <ul style="list-style-type: none"> <li>• Invoke a trigger that starts an action to advance the workflow to the next step.</li> <li>• Resume a workflow or all workflows. The workflow engine can attempt to advance the workflow (or all workflows) to the next step. If there is an error, then the API logs it.</li> <li>• Suspend a workflow.</li> </ul> <p>For more information, see “Workflow Web Services” on page 98 in the <i>Integration Guide</i>.</p>

## Plugins

Renewal jobs use the following plugins. For the complete list, see “Summary of All PolicyCenter Plugins” on page 141 in the *Integration Guide*.

Name	Description
IPolicyRenewalPlugin	Determines whether to start a renewal job. For more information, see “Renewal Plugin” on page 165 in the <i>Integration Guide</i> .
INotificationPlugin	Called by IPolicyRenewalPlugin. Returns the date a notification must be mailed. For more information, see “Notification Plugin” on page 166 in the <i>Integration Guide</i> .
IEffectiveTimePlugin	This plugin determines the initial time component of a policy period's effective/expiration dates (the PeriodStart/PeriodEnd dates) when starting a new job. Renewal is one of the affected jobs. All of this interface's methods have a PolicyPeriod parameter, which can be used to access additional information that might be used to determine the desired time, such as: <ul style="list-style-type: none"> <li>• The line or lines of business</li> <li>• The base jurisdiction</li> <li>• Other revisions</li> </ul> The return value from each method is a Date, with its time component, HH:mm:ss, set to the desired time. The plugin ignores the day component, dd/MM/yyyy. As a convenience, each method returns null to leave the time component set to the default (midnight)
IJobNumberGenPlugin	This plugin generates a new unique job number.
AccountLocationPlugin	For account locations and policy locations, customize how PolicyCenter: <ul style="list-style-type: none"> <li>• Determines that two locations are the same</li> <li>• Clones a location</li> </ul>
IPolicyNumGenPlugin	This plugin returns an identifier for a new period of a policy. The policy number may vary from period to period. Call this plugin for the initial period of a policy as part of a Submission job. You can also call this plugin if you want to generate a new identifier when renewing or rewriting a policy.
IPolicyPeriodDiffPlugin	This plugin does the following: <ul style="list-style-type: none"> <li>• Compares two policy periods, returning a list of DiffItems representing the differences between them. The compareBranches method is called in the following:               <ul style="list-style-type: none"> <li>• Multi-version quote</li> <li>• Comparing PolicyPeriods of a Policy</li> <li>• Comparing Jobs of a Policy</li> <li>• Filters a list of Diff terms that originate from the database.</li> </ul> </li> </ul>
IPolicyPeriodPlugin	This plugin calculates period end dates and creates the job process for that period.
IPolicyPlugin	This plugin contains methods including canStartRenewal which checks to see whether the PolicyPeriod can be renewed.
ITerritoryCodePlugin	This plugin returns the territory associated with the passed-in search criteria. The territory contains a code and a physical place. The plugin requires non-null values for jurisdiction, policy line pattern and effective date.
IVinPlugin	For a VIN (Vehicle Identification Number), the getVehicleInfo method returns the model, make, year, and color of vehicle.

## Renewal Events

Events are changes in PolicyCenter that might be of interest to an external system. The renewal job uses the following events:

- SendRenewalDocuments
- SendNonRenewalDocuments
- SendNotTakenDocuments
- SendNotTaken

- `SendNonRenewal`
- `IssueRenewal`

**See also**

- “Messaging and Events” on page 289 in the *Integration Guide*

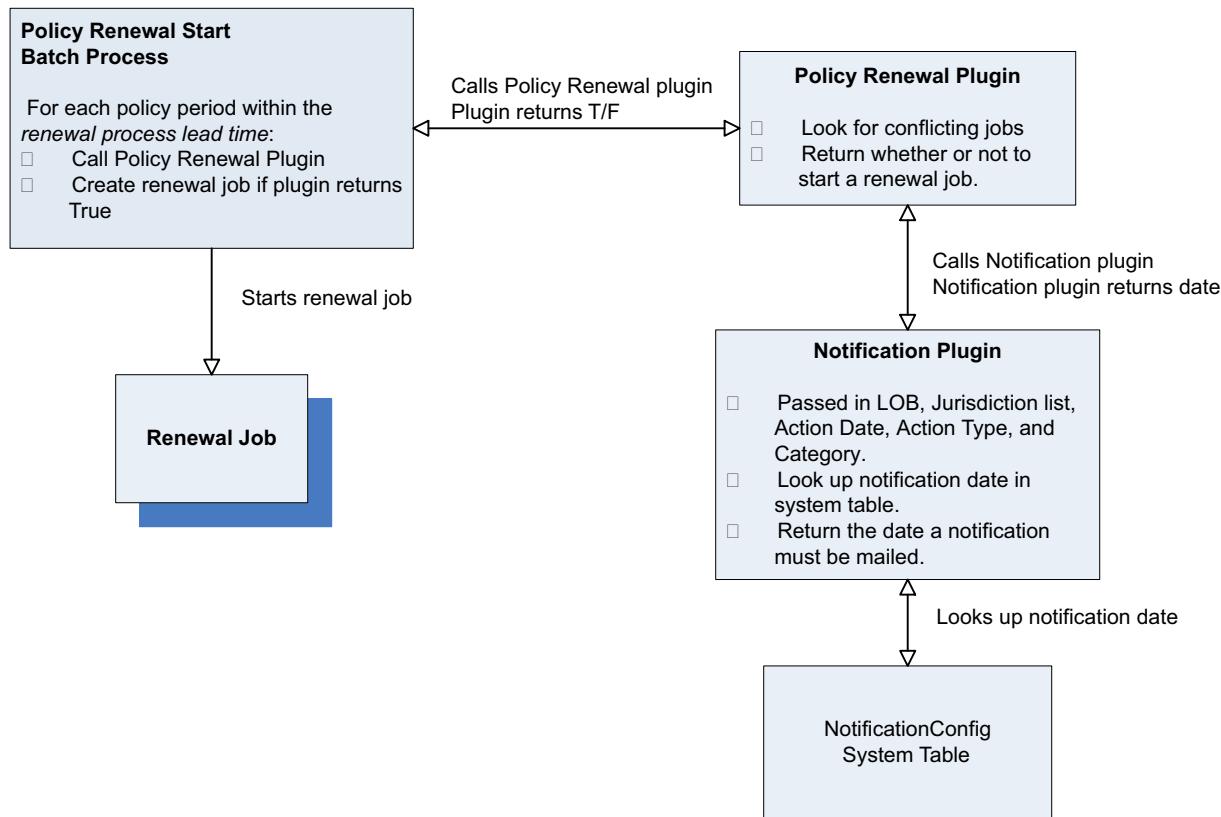
## Configuring Batch Process Renewals

PolicyCenter has a batch process that automatically finds policies that are ready for renewal. In the default configuration, PolicyCenter starts renewals based upon the expiration date, the *renewal process lead time*, line of business, jurisdiction, and time of year. If the expiration date of the policy period falls within the renewal process lead time, then the Renewal plugin calculates whether to start a renewal for the policy. Because the renewal process lead time is checked first, no policy will start automatic renewal sooner than this.

Renewal process lead time is specified in the `RenewalProcessLeadTime` parameter in the `config.xml` file.

**Note:** There are several factors to consider in scheduling the batch process. For more information, see “Factors to Consider in Scheduling the Renewal Batch Process” on page 95 in the *Application Guide*.

The following illustration shows how the Policy Renewal batch process evaluates whether to create renewal jobs for policy periods.



PolicyCenter handles the renewal job process through a series of workflows that monitor its progress and move it to the next step. This series of workflows is the optimal way to begin renewals.

**See also**

- “Creating a Renewal from a Batch Process” on page 103 in the *Application Guide* which includes information on configuring the renewal process lead time
- “Batch Processing” on page 99 in the *System Administration Guide*

## Renewal Plugin Calculation

The Renewal plugin (`IPolicyRenewalPlugin`) determines whether or not to start a renewal on a policy period. You can modify this plugin to incorporate regulatory requirements and carrier practices for starting renewal processing.

In the default configuration, the Renewal plugin determines whether to start a renewal for the policy period in this way. It first determines what the regulations require. Then it adds additional time for company practices. Finally, it adds a delay for concurrent jobs, if any. These factors are described in:

- “Renewal Lead Time in Notification Config System Table” on page 611
- “Example Lead Time by Line of Business” on page 612
- “Example Lead Time by Time of Year” on page 612
- “Delay for a Conflicting Job” on page 612

A renewal starts for a policy period when the current date is greater than or equal to the `RenewalStartDate`. The renewal start can be delayed for a conflicting job.

The `RenewalStartDate` for a policy period is the expiration date of the current period, moved earlier in time by the sum of the following:

Lead time	See
Renewal lead time in Notification Config system table	“Renewal Lead Time in Notification Config System Table” on page 611
Example lead time by line of business	“Example Lead Time by Line of Business” on page 612
Example lead time by time of year	“Example Lead Time by Time of Year” on page 612

If the policy has multiple lines of business, then use the longest lead time determined across all lines of business on the policy. You can configure how to calculate the renewal lead time.

**See also**

- “Renewal Plugin” on page 165 in the *Integration Guide*
- “Notification Plugin” on page 166 in the *Integration Guide*

## Renewal Lead Time in Notification Config System Table

The Notification Config system table is one of the factors in calculating the renewal process lead time.

**Note:** If the policy has multiple lines of business, then default configuration uses the longest lead time determined across all lines of business on the policy.

The Policy Renewal plugin calls the `getMaximumLeadTime` method in the Notification plugin. This method returns the maximum lead time based on the values in the Notification Config table. There are two signatures for this method. The Policy Renewal plugin calls the method with the signature that takes the `Category`, not the `ActionType`. You can find this table by navigating to **System Tables** in Product Designer and opening `notification_configs.xml`.

**See also**

- “Lead Time in NotificationConfig System Table” on page 81 in the *Product Model Guide*

## Example Lead Time by Line of Business

The product lead time by line of business in the default configuration is listed in the following table.

Line of Business	Lead Time
Businessowners	90
Business Auto	30
Commercial Property	60
Commercial Package	90
General Liability	60
Inland Marine	60
Personal Auto	30
Workers Compensation	60

If the policy has multiple lines of business, then the Renewal plugin uses the longest lead time determined across all lines of business on the policy.

You can configure these values in the Renewal plugin.

## Example Lead Time by Time of Year

The following table shows the default configuration of lead time by time of year (date ranges are inclusive):

Date Range	Lead Time
January 1 through January 15	15
January 16 through July 31	0
August 1 through September 15	-10
September 16 through end of year	0

You can configure this in the Renewal plugin. For more information, see “Renewal Plugin” on page 165 in the *Integration Guide*.

## Delay for a Conflicting Job

The batch process does not start a renewal job if there is a conflicting job. A conflicting job is an open policy change or cancellation job. A policy that would normally renew on a given day will wait three additional days to start if there is a conflicting job. After three days, the next run of the batch process starts the renewal, whether or not there is an open conflicting job.

The three day delay is set in the CONCURRENT\_JOB\_DELAY\_DAYS variable which is defined in the Renewal plugin. You can modify how this plugin calculates and uses the delay for a conflicting job.

**Note:** The frequency of the batch process is a factor in when a delayed renewal starts. For example, you have a batch process that runs every Sunday. A delayed renewal that could start on Wednesday will not start until the batch process runs on Sunday, several days later.

## Configuring Explanations in Pre-renewal Directions

In a pre-renewal direction, the user can select a direction to not renew the policy. The user can add explanations to non-renewal directions. These explanations are configured as non-renewal explanation patterns. In Studio, configuration → config → Import → gen contains configuration files related to non-renewal explanation patterns:

File	Description
nonRenewalExplanationPatterns.csv	Definitions for non-renewal explanation patterns.
importfiles.txt	In the default configuration, this file specifies that nonRenewalExplanationPatterns.csv is imported when PolicyCenter is starts.

### See also

- “Creating a Pre-renewal Direction” on page 101 in the *Application Guide*



# Configuring Cancellations

This topic describes how to configure cancellation jobs (cancellation policy transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

- “Configuring Cancellations Overview” on page 615
- “Cancellation Integration” on page 617
- “Calculating the Cancellation Effective Date” on page 619
- “Configuring the Premium Calculation Method” on page 620

**See also**

- “Cancellation Policy Transaction” on page 107 in the *Application Guide*

## Configuring Cancellations Overview

Use Studio to access the necessary files and code to configure the cancellation process to your business needs.

The key configuration files include:

- **Gosu job process class:** `CancellationProcess.gs`. This non-editable (but extendable) class contains the code behind the cancellation job (policy transaction). You can view this class by navigating in Studio to `configuration` → `gsrc` and opening `gw.job.CancellationProcess.gs`. For more information, see “Cancellation Gosu Classes” on page 616.
- **Gosu enhancement file:** `CancellationEnhancement.gsx`. This enhancement allows you to modify or add additional functions and properties to the cancellation job Gosu class. For example, you can modify the code to calculate the `RefundCalcMethod` field in the function `updateRefundCalcMethod`. The `Refund Method` in `CancelPolicyDV` property is set to the value of the `CancellationPolicyPeriod.RefundCalcMethod`. For more information, see “Cancellation Gosu Classes” on page 616.

- **Workflow files** such as `CompleteCancellationWF.1`. This file describes the workflow for completing the asynchronous parts of cancellation. Code in `CancellationProcess.gs` starts the workflow. Clicking **Bind Options** → **Schedule Cancellation** or **Cancel Now** on the **Cancellation** screen calls the code to start the workflow. For more information, see “[Complete Cancellation Workflow](#)” on page 617.
- **Page configuration files (PCF files)** such as `CancellationWizard.pcf`. To view these PCF files, navigate in Studio to **configuration** → **config** → **Page Configuration** → **pcf** → **job** → **cancellation**.
- **Typelists** such as `CancellationSource` or `CalculationMethod`. You can view this typelist by navigating in Studio to **configuration** → **config** → **Metadata** → **Typelist**.
- **System tables** such as `notification_configs.xml`. You can view this table under **System Tables** in Product Designer.

**Note:** Since the cancellation process for any carrier can be configured based on business requirements, all discussions apply to the default application only.

#### See also

For more information about how to control jobs in Gosu and when to use workflows:

- “[Configuring Jobs](#)” on page 589

For more information on configuration:

- “[Cancellation Gosu Classes](#)” on page 616
- “[Complete Cancellation Workflow](#)” on page 617
- “[Cancellation Integration](#)” on page 617
- “[Calculating the Cancellation Effective Date](#)” on page 619
- “[Configuring the Premium Calculation Method](#)” on page 620

## Cancellation Gosu Classes

Gosu classes primarily control cancellations. In Studio, you can configure the Gosu classes by modifying the following types of files:

- “[Cancellation Process Class](#)” on page 616
- “[Cancellation Enhancements](#)” on page 617

### Cancellation Process Class

Navigate in Studio to **configuration** → **gsrc** to view or edit the `gw.job.CancellationProcess.gs` Gosu class which contains the majority of cancellation logic. The `CancellationProcess` class includes functions to start, schedule, quote, rescind, and issue a cancellation. The `CancellationProcess.gs` class extends the `JobProcess` Gosu class. The `JobProcess` class contains functions that allow you to get the job and the policy period.

#### Start Function

Pressing the **Start** button (in `StartCancellation.pcf`) indirectly invokes the `start` function. The **Start** button calls the `Cancellation.startJobAndCommit` function which calls `start`.

#### Modifying the Cancellation Process Class

If you need to change or create new methods in that class, you can edit the class directly or create a subclass (such as `YourCompanyCancellationProcess.gs`).

For more information about job classes and job subclasses, see “[Gosu Classes for Jobs](#)” on page 590.

If you create a subclass, you must modify the job process customization plugin. For more information, see “[Job Process Creation Plugin](#)” on page 152 in the *Integration Guide*.

## Cancellation Enhancements

The cancellation enhancement, `CancellationEnhancement.gsx`, enhances the `Cancellation` entity with additional functionality. You can edit this file.

This enhancement contains functions that do the following:

- Determines which refund calculation functions displays based on the cancellation reason.
- Calculates the earliest and latest effective date of the cancellation.
- Returns the last date of the underwriting period.
- Withdraws later cancellations with the same based on period.

## Complete Cancellation Workflow

**Note:** PolicyCenter uses workflows primarily for automation and coordination of activities between PolicyCenter and external systems and for observing time periods and other automation.

In a cancellation, the job process involves automation and waiting. You wait for a period of time before the cancellation job completes and the policy is canceled. During this time period, the insured may remit payment and the cancellation may be rescinded. The `CompleteCancellationWF.1` workflow manages the processes that are not user driven.

### Complete Cancellation Workflow Steps

The workflow begins when you select **Bind Options** → **Schedule Cancellation** or **Cancel Now** through the PolicyCenter user interface (see “Working with Cancellations” on page 111 in the *Application Guide*).

In the default application, selecting **Schedule Cancellation** or **Cancel Now** in the user interface calls the `scheduleCancellation` function in `CancellationProcess.gs`. After notices are successfully sent through `sendNotices`, the `finishSendNotices` function starts the workflow.

## Cancellation Integration

PolicyCenter integrates with many types of external systems and custom code through a diverse toolbox of services. This topic describes web services and events that the cancellation job uses.

A common integration is with a billing system to start a cancellation for non-payment or to rescind a cancellation after receiving payment.

Another common integration is with a rating engine to calculate the premium amount.

This topic contains the following:

- “[Cancellation Web Services and Plugins](#)” on page 617
- “[Events](#)” on page 618

## Cancellation Web Services and Plugins

The cancellation job has the following web services and plugins.

### Web Services

**The Cancellation API** is a web service for use by external systems to begin, reschedule, rescind, or find a cancellation.

Invoke the `beginCancellation` method to begin a cancellation job. This method does not attempt to automatically push the cancellation through to completion. The method throws an exception if the cancellation cannot be started for any reason.

The `rescheduleCancellation` method allows you to reschedule an cancellation. You must specify the new cancellation effective date. Optionally, you can provide a new reason description. The cancellation can be in draft, quoted, or canceling status.

The `rescindCancellation` method rescinds a cancellation job. The method throws an exception if the rescind cannot be performed for any reason.

#### See also

- “Policy Cancellation and Reinstatement Web Services” on page 105 in the *Integration Guide*

## Plugins

The following plugins might be useful for cancellation.

Name	Description
<code>IBillingPlugin</code>	If you need to integrate with an external billing system, use this plugin to notify an external billing system about billing events.
<code>MessageTransport</code>	Use this plugin to send a message to an external/remote system by using any transport protocol.
<code>IEffectiveTimePlugin</code>	Determines the time of day (since midnight) for a job. Use to calculate the effective time.
<code>IJobNumberGenPlugin</code>	Generates a new, unique job number. Use to generate cancellation job numbers.
<code>IPolicyPlugin</code>	Lets PolicyCenter know whether, based on dynamic calculations on the policy, it is OK to start various jobs. Use to determine if a cancellation job can be started.
<code>INotificationPlugin</code>	Returns cancellation lead time.

#### See also

- “Account and Policy Plugins” on page 149 in the *Integration Guide*

## Events

PolicyCenter generates events for changes in PolicyCenter that might be of interest to an external system. You can also programmatically generate an event. For example, in the `CancellationProcess.gs` class, the `sendNotices` function generates a `SendCancellationNotices` event.

The `Cancellation` entity has the following events:

- `CancellationAdded`
- `CancellationChanged`
- `CancellationRemoved`

The `PolicyPeriod` entity has the following events pertaining to cancellation:

- `SendCancellationNotices` – Message to an external system to send cancellation notices.
- `IssueCancellation` – Message to an external System of Record (SOR) to send cancellation notices.
- `SendRescindNotices` – Message to an external system to send rescind cancellation notices.

#### See also

- “List of Messaging Events in PolicyCenter” on page 309 in the *Integration Guide*

## Calculating the Cancellation Effective Date

PolicyCenter calculates the earliest cancellation effective date, based on cancellation reason, jurisdiction, line of business, and other factors. Methods in `CancellationEnhancement.gsx` calculate the date. In some cases, the calculation calls the Notification plugin which references the `notification_configs.xml` system table. The `notification_configs.xml` system table is where you specify the cancellation lead time by jurisdiction, line of business, and cancellation reason. You can view this table by navigating to **System Tables** in Product Designer.

### Calculating the Cancellation Effective Date if the Source is Insured

If the **Source is Insured**, return the current date. However, if the reason is that the policy is not taken, return the date that the policy went into effect.

### Calculating the Cancellation Effective Date if the Source is Carrier

This topic describes how PolicyCenter calculates the cancellation effective if the **Source is Carrier**.

**Note:** The cancellation lead time returned by the `notification_config.xml` system table is described in “Lead Time in NotificationConfig System Table” on page 81 in the *Product Model Guide*. This topic also describes how the application determines the lead time.

1. If the cancellation reason is equal to **Policy rewritten or replaced (flat cancel)**, then the cancellation effective date is equal to starting date of the policy period. You are done.
2. Else, determine if the policy is in the underwriting period.
  - a. Calculate the number of the days (*xDays*) between the Policy Effective Date and the current date.
  - b. The Notification plugin retrieves the underwriting period from the `notification_configs.xml` system table. The underwriting period is specified in rows with `ActionType` of `Underwriting Period`. The underwriting period is the value in the `LeadTime` column.  
If *xDays* is less than or equal to the `LeadTime`, then the policy is in the underwriting period.  
If the policy is in the underwriting period, the application will use underwriting period action types and categories in the `notification_configs.xml` system table to determine the cancellation lead time.
3. The Notification plugin retrieves the maximum lead time required for notification from the `notification_configs.xml` system table. The plugin retrieves the maximum lead time based on the cancellation reason, whether the policy is in the underwriting period, the policy jurisdiction, and the line of business.

Cancellation reason	Category	ActionType
Fraud	Cancellation	<code>fraudcancel</code>
	UW Period Cancellation	<code>uwperiodfraudcancel</code>
Non payment	Cancellation	<code>nonpaycancel</code>
	UW Period Cancellation	<code>uwperiodnonpaycancel</code>
Any other reason	Cancellation	<code>othercancel</code>
	UW Period Cancellation	<code>uwothercancel</code>

**Note:** If a policy includes multiple jurisdictions, there can be multiple matching rows. A policy that includes more than one line of business (such as a commercial package policy) can include both multiple jurisdictions and lines of business. In this case, the Notification plugin retrieves the maximum lead time. Therefore, the plugin returns the longest lead time specified for any of the jurisdiction and line of business combinations in the policy.

The cancellation effective date is equal to the current date plus the `LeadTime` for the matching row.

## Configuring the Premium Calculation Method

You cannot add or change the refund method (pro rata, short rate, and flat) except to change the calculation of penalty in short rate. In short rate, a rating engine calculates the amount of penalty. The default configuration contains rating system plugins. You can modify the plugin algorithms or integrate your own rating system to work with PolicyCenter.

# Configuring Policy Change

This topic describes how to configure policy change jobs (policy change transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

- “Configuring Policy Change Overview” on page 621
- “Policy Change Integration” on page 623

**See also**

- “Policy Change Transaction” on page 115 in the *Application Guide*

## Configuring Policy Change Overview

Gosu classes primarily control policy changes. In Studio, navigate to **configuration** → **gsrc** to view or edit the `gw.job.PolicyChangeProcess.gs` Gosu class which contains most of the policy change code.

In a policy change job, the `JobProcess.gs` class, the `QuoteProcess.gs` class, and the `PolicyChangeProcess.gs` class contain code for the main job actions. The default application spreads the policy change code among multiple classes for better organization and use of code. For example, changing the quote logic can be done in one location (`QuoteProcess.gs`). You can modify the quote logic in one location and many types of jobs can access it.

**Note:** In the default configuration, the policy change job has no workflows because it has no background processes. PolicyCenter uses workflows only for background processes.

This topic includes the following:

- “Policy Change Process Class” on page 622
- “Modifying the Policy Change Process Class” on page 623

## Policy Change Process Class

The `PolicyChangeProcess` class contains method for handling policy change jobs. The following table summarizes some of the methods contained in the `PolicyChangeProcess.gs` class.

**Note:** To see the complete list of methods, view the `gw.job.PolicyChangeProcess` class in Studio.

Method	Description
<b>Starting a policy change</b>	
<code>start</code>	This method starts the job and automatically assigns a requestor and producer.
<code>startAutomatic</code>	This method calls three methods which start the job, request a quote, and bind the policy. The <code>PolicyChangeAPI</code> calls this method.
<b>Binding a policy change</b>	
<code>canBind</code>	<code>canBind</code> – Checks that the policy can be bound. Some of the checks include: <ul style="list-style-type: none"> <li>• Has the policy been quoted?</li> <li>• Does the user have the correct permissions to bind the policy?</li> </ul>
<code>bind</code>	Begins the binding process for a <code>PolicyPeriod</code> .  This method verifies if the branch can be bound. For multi-version jobs, there is only one selected branch on the job which is specified by <code>Job.SelectedVersion</code> . That branch must have a valid quote. If the branch can be bound, then <code>bind</code> starts the binding process and sets the policy status to <code>Binding</code> .  This method ensures that there is a producer of service on the policy.  This method also calls out to the <code>FormInferenceEngine</code> , which is the main integration point for forms.  Lastly, the <code>startBinding</code> method has commented out code to add an <code>IssuePolicyChange</code> event.
<code>finishBinding</code>	Completes the policy change job. This method also checks new conditions and sends billing information through the <code>IBillingSystemPlugin</code> .
<code>failBinding</code>	If binding fails, then the policy can be edited again by an underwriter.
<b>Editing the effective date</b>	
<code>canStartChangeEditEffectiveDate</code>	Check to see if <code>Actions → Edit → Effective Date</code> is available.
<code>canFinishChangeEditEffectiveDate</code>	Check to see if the current policy change can change its effective date to the input parameter value.
<code>changeEditEffectiveDate</code>	<code>PolicyCenter</code> invokes this method when you select <code>Effective Date</code> in the <code>Actions</code> menu.  If <code>canStartChangeEditEffectiveDate</code> is false, then this method is not called.  This method calls <code>canFinishChangeEditEffectiveDate</code> to verify that the change is valid before applying it.
<b>Handling future bound jobs</b>	
<code>applyChangesToFutureBoundRenewal</code>	Applies changes from this policy change to a renewal that is bound in the future.
<code>applyChangesToFutureUnboundRenewal</code>	Applies changes from this policy change to a renewal that is unbound in the future.
<b>Flags</b>	
<code>canRequestQuote</code>	Indicates whether quoting can be initiated on the <code>PolicyPeriod</code> .
<code>canWithdraw</code>	Checks to see if a branch can be withdrawn.

The `PolicyChangeProcess` class also identifies and raises underwriting issues with the policy, using the `UWIssueType` system table to identify the types of issues to raise.

### See also

- “Configuring Underwriting Authority” on page 469

- “Adding a New Checking Set” on page 475
- “Underwriting Issue Type System Table” on page 489

## Modifying the Policy Change Process Class

If you need to change or create new methods in that class, you can edit the class directly or create a subclass (such as `YourCompanyPolicyChangeProcess.gs`).

For more information about job classes and job subclasses, see “Gosu Classes for Jobs” on page 590.

If you create a subclass, you must modify the job process customization plugin. For more information, see “Job Process Creation Plugin” on page 152 in the *Integration Guide*.

## Policy Change Integration

PolicyCenter integrates with many types of external systems through a diverse set of services that can link PolicyCenter with custom code and external systems. This topic describes APIs and events that are relevant to the policy change job (policy transaction).

### Policy Change Web Services and Plugins

The policy change job has the following web services and plugins.

#### Web Services

The `PolicyChangeAPI` includes two methods to start policy change jobs:

- The `startAutomaticPolicyChange` method starts a policy change job for a policy (specified by policy number) and an effective date. It returns the job number of the new job (its `JobNumber` property). In the default configuration, this method does not make any changes to the policy data. You can add your own methods to make changes to the data.
- The `startManualPolicyChange` method has the same arguments and return values, but does not attempt to bind and quote automatically. The policy change starts and waits in draft mode for the user to quote and finish it manually.

#### Plugins

The policy change job has the following plugins.

##### **IPolicyPlugin**

The `canStartPolicyChange` method returns an error message if a change cannot be started for a policy and an effective date. Changes that are started internally or externally call this plugin.

For more information, see “Policy Plugin” on page 158 in the *Integration Guide*.

##### **IPolicyPeriodPlugin**

The `postCreateNewBranchForChangeEditEffectiveDate` method is called to handle a change to the effective date of a job such as a policy change. In the default configuration, policy change is the only job that calls this method.

## Policy Change Events

In PolicyCenter, events are changes (such as a new policy change) that might be of interest to an external system. In the `bind` method of the `PolicyChangeProcess.gs` class, you can generate the `IssuePolicyChange` event. In the default configuration, this code is commented out. The code notifies an external system to issue a policy change.

### See also

- “Messaging and Events” on page 289 in the *Integration Guide*

# Configuring Reinstatement

This topic describes how to configure reinstatement jobs (reinstatement policy transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

- “Configuring Reinstatement Overview” on page 625
- “Reinstatement Integration” on page 626

**See also**

- “Reinstatement Policy Transactions” on page 125 in the *Application Guide*

## Configuring Reinstatement Overview

Use Guidewire Studio to access the files and code to configure the reinstatement process to your business needs.

Key files include:

- **Gosu job process class** – `ReinstateProcess.gsx`. This class contains the code behind the reinstatement job. You can view or edit this class in Studio by navigating to `configuration → gsrc` and selecting `gw.job.ReinstateProcess.gsx`. For more information, see “Reinstate Process Gosu Class” on page 626
- **Gosu enhancement file** – `ReinstateEnhancement.gsx`. This enhancement allows you to modify or add additional functions and properties to the reinstatement job Gosu class. For example, this enhancement class has just one function which is called when the reinstatement job finishes. This function restarts any withdrawn renewals on the policy.
- **Page configuration files (PCF files)** such as `ReinstateWizard.pcf`. To view these PCF files, navigate in Studio to `configuration → config → Page Configuration → pcf → job → reinstate`.

- **TypeLists** such as `ReinstateCode`. You can view this typeList by navigating in Studio to **configuration** → **config** → **Metadata** → **TypeList**.

**Note:** Since the reinstatement process for any carrier can be configured based on business requirements, all discussions apply only to the default application.

To learn more about how jobs work in Gosu and how to use workflows, see “Configuring Jobs” on page 589.

## Reinstatement Process Gosu Class

Gosu classes primarily control reinstatements. In Studio, navigate to **configuration** → **gsrc** to view the `gw.job.ReinstatementProcess.gs` Gosu class which contains most of the reinstatement code.

The `ReinstatementProcess.gs` contains functions for reinstatement. The `JobProcess.gs` contains general functions used by different types of jobs.

The `ReinstatementProcess.gs` class extends the `JobProcess` Gosu class. The `JobProcess` class contains functions that allow you to get the job and the policy period. It cancels open activities on the reinstatement and auto assigns the user role. It also checks to see if the job can be expired or withdrawn, and checks if the job can be started as new.

This class identifies and raises underwriting issues with the policy, using the `UWIssueType` system table to identify the types of issues to raise.

Methods in `ReinstatementProcess.gs` start the job as a draft, then automatically assign the producer and underwriter roles to the job.

**Note:** The reinstatement job has no workflows because it has no background processes. PolicyCenter uses workflows only for background processes.

### See also

- “Configuring Underwriting Authority” on page 469
- “Underwriting Issue Type System Table” on page 489

## Modifying the Reinstatement Process Class

If you need to change or create new methods in that class, you can edit the class directly or create a subclass (such as `YourCompanyReinstatementProcess.gs`).

For more information about job classes and job subclasses, see “Gosu Classes for Jobs” on page 590.

If you create a subclass, you must modify the job process customization plugin. For more information, see “Job Process Creation Plugin” on page 152 in the *Integration Guide*.

## Reinstatement Integration

PolicyCenter integrates with many types of external systems through a diverse toolbox of services that can link PolicyCenter with custom code and external systems. This topic describes web services and events that the reinstatement job uses.

## Reinstatement Web Services and Plugins

This topic describes the web services and plugins for reinstatement.

### Web Services

The `ReinstateAPI` is a web service to start a reinstatement from an external system.

The `ReinstatementAPI.beginReinstatement` function can be invoked to begin a reinstatement job. It does not attempt to automatically push the reinstatement job through to completion. The `policyNumber` and `reinstateCode` arguments identify which policy is to be reinstated and the reason for the reinstatement. The function throws an exception if the reinstatement cannot be started for any reason.

**See also**

- “Policy Cancellation and Reinstatement Web Services” on page 105 in the *Integration Guide*

## Plugins

The following plugins may be useful to integrate with reinstatement.

Name	Description
<code>IBillingPlugin</code>	If you need to integrate with an external billing system, use this plugin to notify an external billing system about billing events.
<code>MessageTransport</code>	Use this plugin to send a message to an external/remote system by using any transport protocol.
<code>IEffectiveTimePlugin</code>	Determines the time of day (since midnight) for a job. Use to calculate the effective time.
<code>IJobNumberGenPlugin</code>	Generates a new, unique job number. Use to generate reinstatement job numbers.
<code>IPolicyPlugin</code>	Lets PolicyCenter know whether to start various jobs, based on dynamic calculations on the policy. Use to determine when a reinstatement job can be started.

**See also**

- “Account and Policy Plugins” on page 149 in the *Integration Guide*

## Events

PolicyCenter generates events for changes that might be of interest to an external system. You can also programmatically generate an event. For example, in the `ReinstatementProcess.gs` class, the function `startReinstate` generates an `IssueReinstatement` event.

The `Reinstatement` entity has the following events:

- `ReinstatementAdded`
- `ReinstatementChanged`
- `ReinstatementRemoved`

The `PolicyPeriod` entity has the following events pertaining to reinstatement:

- `IssueReinstatement` – a message that notifies an external system to issue a reinstatement for a particular policy period.

**See also**

- “List of Messaging Events in PolicyCenter” on page 309 in the *Integration Guide*





## chapter 50

# Configuring Rewrite

This topic describes how to configure rewrite jobs (rewrite policy transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

- “Configuring Rewrite Overview” on page 629
- “Rewrite Integration” on page 630

**See also**

- “Rewrite Policy Transactions” on page 129 in the *Application Guide*
- “Selecting the Underwriting Company through Segmentation” on page 595

## Configuring Rewrite Overview

Use Studio to access the necessary files and code to configure the rewrite job (policy transaction) to your business needs.

Key files include:

- **Gosu job process class** – RewriteProcess.gs. This class contains the code behind the rewrite job. You can view or edit this class by navigating in Studio to **configuration** → **gsrc**, and selecting **gw.job.RewriteProcess.gs**. For more information, see “Rewrite Process Gosu Class” on page 630.
- **Page configuration files (PCF files)** such as RewriteWizard.pcf. To view these PCF files, navigate in Studio to **configuration** → **config** → **Page Configuration** → **pcf** → **job** → **rewrite**.
- **Typelists** such as RewriteType. You can view this typelist by navigating in Studio to **configuration** → **config** → **Metadata** → **Typelist**.

**Note:** Since the rewrite process for any carrier can be configured based on business requirements, all discussions apply only to the default application.

**See also**

- “Configuring Jobs” on page 589 to learn more about how jobs work in Gosu and how to use workflows

## Rewrite Process Gosu Class

Gosu classes primarily control the rewrite process. In Studio, navigate to **configuration** → **gsrc** and open the `gw.job.RewriteProcess.gs` Gosu class which contains most of the rewrite code.

The `RewriteProcess.gs` class extends the `JobProcess` Gosu class. The `JobProcess` class contains functions that allow you to get the job and the policy period. It cancels open activities on the rewrite and auto assigns the user role. It also checks to see if the job can be expired, or withdrawn, and checks if the job can be started as new.

Functions in `RewriteProcess.gs` start the rewrite job, request a quote, and rewrite the policy period.

**Note:** The rewrite job has no workflows because it has no background processes. PolicyCenter uses workflows only for background processes.

### Modifying the Rewrite Process Gosu Class

If you need to change or create new methods in that class, you can edit the class directly or create a subclass (such as `YourCompanyIssuanceProcess.gs`).

For more information about job classes and job subclasses, see “Gosu Classes for Jobs” on page 590.

If you create a subclass, you must modify the job process customization plugin. For more information, see “Job Process Creation Plugin” on page 152 in the *Integration Guide*.

## Rewrite Integration

PolicyCenter integrates with many types of external systems through a diverse toolbox of services that can link PolicyCenter with custom code and external systems. This topic describes events for the rewrite job (policy transaction). The rewrite job has no web services.

### Events

PolicyCenter generates events for changes in PolicyCenter that might be of interest to an external system. You can also programmatically generate an event. For example, in the `RewriteProcess.gs` class, the `rewrite` function generates an `IssueRewrite` event (by calling the `startBinding` function).

The `Rewrite` entity has the following events:

- `RewriteAdded`
- `RewriteChanged`
- `RewriteRemoved`

The `PolicyPeriod` entity has the following events pertaining to rewrite:

- `IssueRewrite` – A message that notifies an external system to issue a rewrite for a particular policy period.

### See also

- “List of Messaging Events in PolicyCenter” on page 309 in the *Integration Guide*

# Configuring Rewrite New Account

This topic describes how to configure rewrite new account jobs (rewrite new account policy transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

- “Configuring the Rewrite New Account Job Overview” on page 631

**See also**

- “Rewrite New Account Policy Transaction” on page 135 in the *Application Guide*

## Configuring the Rewrite New Account Job Overview

This topic describes how to configure the rewrite new account job (policy transaction).

### Rewrite New Account Job

The main PCF file for the rewrite new account is the `RewriteNewAccountWizard`.

The primary Gosu class for this job is `gw.job.RewriteNewAccountProcess`. The main methods in this class are:

- `start`
- `rewriteNewAccount`
- `finishRewriteNewAccount`

If you need to change or add methods to the rewrite new account process, you can edit the job class directly or create a subclass (such as `YourCompanyRewriteNewAccountProcess.gs`).

For more information about job classes and job subclasses, see “Gosu Classes for Jobs” on page 590.

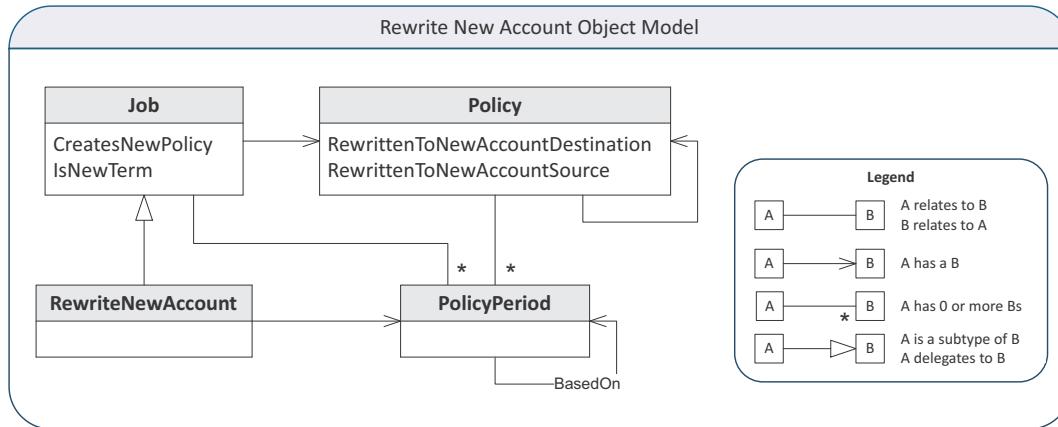
If you create a subclass, you must modify the job process customization plugin. For more information, see “Job Process Creation Plugin” on page 152 in the *Integration Guide*.

## Copying Contacts and Locations

When a rewrite new account job starts, it copies contacts and locations from the source policy account to the target account. For example, the source policy period references the driver, John Doe. The job checks for a corresponding **AccountContact** and **Driver** for John Doe on the target account and creates these entities if they do not exist. In the rewritten policy, the corresponding policy driver points to the copied account level driver.

## Rewrite New Account Object Model

This topic describes entities and fields related to the rewrite new account job (policy transaction).



### Rewrite New Account Entity

The **RewriteNewAccount** entity represents the rewrite new account job. This entity has a **PolicyPeriod** field which returns the policy period associated with the job.

The **PolicyPeriod** entity associated with the **RewriteNewAccount** entity has a **BasedOn** field that returns the policy period on which the source policy is based.

### Policy Entity

The **Policy** entity has the following fields related to the rewrite new account job:

Field	Description
<b>RewrittenToNewAccountDestination</b>	If this policy was rewritten to a new account, returns the destination policy.
<b>RewrittenToNewAccountSource</b>	If this policy was rewritten to a new account, returns the source policy.

### Job Entity

In a rewrite new account job, the **Job** entity has the following values:

Field	Description
<b>CreatesNewPolicy</b>	Value is True if this job creates a new policy with a new policy number. This value is True for rewrite new account jobs.
<b>IsNewTerm</b>	Value is True if this job creates a new policy term. This value is True for rewrite new account jobs.

## Rewrite New Account History Events

The rewrite new account job (policy transaction) creates the following history events:

History event	Description
rewr_new_acct_created	This event appears on the source policy term when the rewrite new account job starts.
rewr_new_acct_bound	This event appears on the target policy term when the job is bound.



# Configuring Premium Audit

This topic describes how to configure premium audit jobs (premium audit policy transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

- “Configuring Premium Audit Overview” on page 635
- “Final Audit Integration” on page 638

**See also**

- “Premium Audit Policy Transaction” on page 143 in the *Application Guide*

## Configuring Premium Audit Overview

Use Guidewire Studio to access the necessary files and code to configure the audit job (policy transaction) to your business needs.

Key files include:

- **Gosu job process class:** `AuditProcess.gs`. This class contains the code behind the final audit job. You can view or edit this class in Studio by navigating to `configuration → gsrc` and opening `gw.job.AuditProcess.gs` class. For more information, see “Audit Gosu Classes” on page 637.
- **Gosu enhancement files:**
  - `AuditInformationEnhancement.gsx` – This enhancement allows you to modify or add additional functions and properties to the final audit job Gosu class.
  - `AuditAssignmentEnhancement.gsx` – This enhancement assigns the premium auditor to the job if the premium auditor was not already assigned. You can view or edit this class in Studio by navigating to `configuration → gsrc` and opening `gw.assignment.AuditAssignmentEnhancement.gsx`.

- **Rule Sets** – The Reporting Trend Analysis → Ratio Out of Range rule set checks to see if the reporting trend analysis ratio is within an acceptable range. In the default configuration, the acceptable range is from 90% to 110%. If the ratio is outside the acceptable range and the number of reporting days is greater than 60, the code creates a `RatioOutOfRange` activity and assigns it to an underwriter. For information about reporting trend analysis, see “Premium Report Trend Analysis” on page 147 in the *Application Guide*.
- **Page configuration files (PCF files)** such as `AuditWizard.pcf`. To view these PCF files, navigate in Studio to `configuration` → `config` → `Page Configuration` → `pcf` → `job` → `audit`.
- **Typelists** such as `AuditMethod`. Typelists are in the `configuration` → `config` → `Metadata` → `Typelist` folder in Studio.

**IMPORTANT** Since the audit process for any carrier can be configured based on business requirements, all discussions apply to the default application only.

#### See also

- “Configuring Jobs” on page 589 to learn more about how jobs work in Gosu and how to use workflows
- “Adding Premium Audit to a Line of Business” on page 189 in the *Product Model Guide*

## Audit Schedules

PolicyCenter uses the audit schedules during the creation of an audit or premium report. The audit schedule specifies the start date, the due date, and the method (physical, voluntary, or by phone). PolicyCenter calculates the start date and the due date based on the period end date. The premium report schedule has other fields including frequency and default deposit.

### Final Audit Schedules

The default application comes with a variety of final audit schedules including schedules for cancellations audits and expirations audits. You can view these schedules in Product Designer by navigating to **Audit Schedules**.

In Product Designer, you can view **Expiration Audit by Physical**, which provides an example of a final audit schedule. The audit is set to start 30 calendar days before the audit period end date. It is due 45 calendar days after audit period end date. If either the due or start date falls on a non-business day, it is delayed until the next business day.

The default final audit schedules are:

- **Cancellation Audit by Phone** for all cancellation final audits
- **Expiration Audit by Physical** for all non-cancellation final audits

The `IAuditSchedulePatternSelectorPlugin` determines the default final audit schedule. You can add additional audit schedules and modify the audit schedule pattern selector plugin code. For example, you can modify the plugin to auto-assign the schedule based on criteria such as policy premium.

### Premium Report Schedules

Premium report scheduled items may be configured to utilize policy months or calendar months. In the default application, the premium report schedules are:

- **Monthly Reports by calendar months, exclude last month**
- **Monthly Reports by policy month, exclude last month**
- **Quarterly Reports by calendar quarter for full policy term**
- **Quarterly Reports by calendar quarter, exclude last quarter**
- **Quarterly Reports by policy quarter, exclude last quarter**

If you select policy months, the start and end day of the month is based on the policy's effective date. The following table shows reports for an audit schedule frequency of **Monthly reports by policy months**.

Policy effective date 3/28/09	Policy effective date 03/10/09
03/28/09 to 04/28/09	03/10/09 to 04/10/09
04/28/09 to 05/28/09	04/10/09 to 05/10/09
05/28/09 to 06/28/09	05/10/09 to 06/10/09
...	...

If you select calendar months, an additional audit schedule field comes into play. This audit schedule field is the *round to next month* field. This field allows the carrier to control for a first report that is very short. If the policy effective date is after the *round to* date, then the first report covers the first short month and the second month. Subsequent periods correspond to calendar months. The following table shows reports for an audit schedule of **Monthly reports by calendar months with round to next month on day 15**.

Policy effective date 3/28/09	Policy effective date 03/10/09
03/28/09 to 05/01/09	03/10/09 to 04/01/09
05/01/09 to 06/01/09	04/01/09 to 05/01/09
06/01/09 to 07/01/09	05/01/09 to 06/01/09
...	...

In Product Designer, you can view **Monthly Reports by calendar month, excluding last month**, which provides an example of a premium report schedule. The schedule specifies that the audit method is voluntary and has a 10% deposit. In the schedule details, the minimum audit period length is set to 15 days. The last audit period is excluded. In the period details, the audit is set to start five calendar days before the audit period end date. The audit is due 15 calendar days after the audit period end date.

## Audit Gosu Classes

Gosu classes primarily control final audits. In Studio, navigate to `configuration → gsrc` and open the `gw.job.AuditProcess.gs` Gosu class which contains most of the audit code.

- The `AuditProcess.gs` contains methods for audit. The `JobProcess.gs` contains general methods used by different types of jobs.

The `AuditProcess.gs` class extends the `JobProcess` Gosu class. The `JobProcess` class contains methods that allow you to get the job and the policy period. It auto assigns the user role, and checks to see if the job can be expired, or withdrawn, and checks if the job can be started as new.

Methods in `AuditProcess.gs` start the job as a draft, and verify the conditions for which the audit can be edited. For example, does the user have edit permissions? Is the job not complete? A method checks to see if an audit package can be created. The `complete` method moves a policy period into the `AuditComplete` status, marking the audit as complete. From then on the policy period is read-only, and any new work on the audit must be done by revising the audit. Lastly, since an audit can be waived through the `waive` method, it locks the audit and it too cannot be edited.

- The `AuditInformationEnhancement.gsx` file contains methods that check to see if the audit is revisable. An audit is revisable if it is completed and was not waived, reversed, or already revised.
- The `PolicyPeriodAuditEnhancement.gsx` file contains methods for the policy period that are related audit. This enhancement contains methods to schedule and remove a final audit from a policy period.

This enhancement also contains a `reverseFinalAudit` method which reverses an audit. This method makes a call to the billing system to reverse charges related to the final audit. This method also sets the `AuditInformation.ReversalDate` to the current date. You can see which audits have been reversed by examining the `ReversalDate`.

PolicyCenter indicates that a policy has been billed by making an integration call to the billing system and calling the `PolicyPeriod.markBilled` function. See “Final Audit Integration” on page 638 for more information.

**Note:** The final audit job has no workflows because it has no background processes. PolicyCenter uses workflows only for background processes.

### Modifying the Audit Process Gosu Class

If you need to change or create new methods in the `AuditProcess` Gosu class, you can edit the class directly or create a subclass (such as `YourCompanyAuditProcess.gs`).

For more information about job classes and job subclasses, see “Configuring Jobs” on page 589.

If you create a subclass, you must modify the job process customization plugin. For more information, see “Job Process Creation Plugin” on page 152 in the *Integration Guide*.

## Final Audit Integration

PolicyCenter integrates with many types of external systems through a diverse toolbox of services that can link PolicyCenter with custom code and external systems. Your integration may need to consider integrating activities, workflows, messages (notes or email) to these external systems. This topic describes web services and events for the audit job (policy transaction).

### Integration with BillingCenter

The base configuration of PolicyCenter contains an integration with Guidewire BillingCenter. For information on activating, using, and configuring this integration, see “Billing System Integration” on page 717 in the *Application Guide*.

### Audit Web Services and Plugins

This topic describes APIs and plugins for audit.

#### Web Services

There are no audit web services.

#### Plugins

The following plugins may be useful to integrate with audit.

Name	Description
<code>IAuditSchedulePatternSelectorPlugin</code>	This plugin sets the default audit schedule for final audits. In the base configuration, cancellation final audits are scheduled by phone and expiration final audits are scheduled physically.  If the audit method is dependent on the premium value, you can configure it in this plugin.
<code>IBillingSystemPlugin</code>	If you need to integrate with an external billing system, use this plugin to notify an external billing system about billing events.

For further information, see “Account and Policy Plugins” on page 149 in the *Integration Guide*.

## Events

PolicyCenter generates events for changes that might be of interest to an external system. You can also programmatically generate an event.

There are no events for audit.



# Configuring Side-by-side Quoting

This topic describes how to configure side-by-side quoting for jobs (policy transactions).

**Note:** In PolicyCenter, the user interface uses the term *policy transaction* to refer to submissions, policy changes, and other policy transactions. Policy transactions are implemented as jobs in the data model, and referred to as jobs in PCF files, Gosu classes, and other configuration files. Therefore, the configuration documentation refers to policy transactions as jobs.

This topic includes:

- “Changing the Maximum Number of Versions in Side-by-side Quoting” on page 641
- “Marking a Job as Side-by-side” on page 642
- “Side-by-side Quoting Object Model” on page 642
- “Configuring the Side-by-side Quoting Screen” on page 642
- “Copying Base Data for Side-by-side Quoting” on page 644
- “Excluding Side-by-side Data from Base Data” on page 647
- “Configuring Quote All to Ignore Validation Warnings” on page 654

**See also**

- “Side-by-side Quoting” on page 161 in the *Application Guide*
- “Multiple Revision Jobs and the Job Selected Branch Property” on page 595

## Changing the Maximum Number of Versions in Side-by-side Quoting

In the default configuration, the maximum number of side-by-side quotes is four. You can change the maximum number of side-by-side quotes for each job (policy transaction) type. In `config.xml`, the configuration parameters are:

- `RenewalMaxSideBySideQuotes`
- `SubmissionMaxSideBySideQuotes`
- `PolicyChangeMaxSideBySideQuotes`

On the Job entity, the derived property `MaxNumberOfSideBySideQuotes` returns the maximum number of side-by-side quotes specified in `config.xml` for that job.

#### See also

- “Side-by-Side Quoting Parameters” on page 77

## Marking a Job as Side-by-side

Setting the `SideBySide` bit on the Job entity marks a job (policy transaction) as side-by-side.

## Side-by-side Quoting Object Model

This topic describes the object model for side-by-side quoting.

### Job Entity

The Job entity has the following fields related to side-by-side quoting:

Field	Description
<code>SideBySide</code>	Specifies whether this job has side-by-side quoting.
<code>SelectedVersion</code>	This foreign key points to the <code>PolicyPeriod</code> of the selected version.
<code>MaxNumberOfSideBySideQuotes</code>	This derived property returns the maximum number of side-by-side quotes specified in <code>config.xml</code> for the job type.

### Base Data Entities

The `gw.api.logicalmatch.LogicalMatcher` interface provides methods which determine if two things are logically equivalent. There are other interfaces which extend the `LogicalMatcher` interface, such as the `EffDatedLogicalMatcher` interface. Entities which implement this interface use the `isLogicalMatcherUntyped` method to determine whether the entities are equivalent. If this method returns `true` then the two entities are logically equivalent in side-by-side periods.

If you add a new entity that you would like to be base data, add a matcher that compares two entities and determines whether the entities are equivalent.

The `gw.lob.pa.PersonalVehicleMatcher` is an example matcher for the `PersonalVehicle` entity. The entity definition shows that the entity implements the `LogicalMatcher` interface. In Studio, press `CTRL+SHIFT+N` and enter `PersonalVehicle.eti`. Then double-click the search result to open the file in the editor. View the `implementsInterface` row with the `Primary Value` equal to `gw.api.logicalmatch.EffDatedLogicalMatcher`. This interface is implemented by the `gw.lob.pa.PersonalVehicleMatcher` class (`Secondary Value`). In general, the matcher is in the same package as the entity for which it provides matching.

## Configuring the Side-by-side Quoting Screen

The PCF file for the **Side-by-Side Quoting** screen is `SideBySideScreen.pcf`. In Studio, press `CTRL+SHIFT+N` and enter the filename. Then double-click the search result to open the file in the editor.

You can configure PolicyCenter to display base data fields on the **Side-by-Side Quoting** screen. For example, you can add widgets for base data fields in the columns for each side-by-side version.

**IMPORTANT** Guidewire requires that base data entities or fields on the **Side-by-Side Quoting** screen not be editable in more than one place on a given screen.

Placing an editable widget for a base data field in the columns replicated for each version is a violation of this requirement. This requirement applies to fields that are implicitly base data, such as contact or location information that can be synchronized.

## Posting Changes to the Server

For performance reasons, the **Side-by-Side Quoting** screen in the base configuration posts changes to the server only as needed.

The default configuration follows these rules for posting changes to the server on the **Side-by-Side Quoting** screen:

- **Quoted** – If the period is in the quoted state, PolicyCenter posts when the user edits any field on that period. In the **Side-by-Side Quoting** screen, PolicyCenter requires the post when you edit a quoted period because PolicyCenter must open that period for edit. (The user does not explicitly open the period for edit.) Opening the period for edit invalidates the premiums, and moves the policy period from quoted to draft status.
- **Not quoted** – If the edited period is not in the quoted state,
  - A change to a widget, such as a check box, that adds or removes a coverage triggers a post to the server. The post ensures that PolicyCenter reevaluates the availability logic in the product model.
  - An edit to a coverage term value does not trigger a post.In personal auto, the one exception to this rule is the **PALiability** coverage and its coverage terms. The **PALiability** coverage has dependencies with other coverages in the policy period, therefore a post is necessary.

Regardless of the policy period status:

- Clicking **Quote All** or **New Version** triggers a post and a page refresh.
- Clicking **Use Defaults** for an offering triggers a post and a page refresh.

## Disabling Post on Enter

Most changes on the **Side-by-Side** screen post changes back to PolicyCenter. However, posting is not necessary for certain types of changes. By default, post on change is not selected for PCF elements.

### To programmatically determine whether to post on change

1. In Studio, open a PCF file, such as `SideBySideCovTermInputSet.bit.pcf`.
2. Select a PCF element, such as the `BooleanRadioInput`.
3. In **Properties**, select the **PostOnChange** tab, then select **Enable targeted Post On Change**.
4. In the `disablePostOnEnter` property, enter code to set this property.

When enabled, PolicyCenter checks the value of `disablePostOnEnter` to determine whether to post changes.

## Configuring Initial Behavior of the Side-by-side Quoting Screen

In personal auto, when you view the **Side-by-Side Quoting** screen, PolicyCenter creates two additional periods for a total of three side-by-side periods. The new periods are copies of the initial period. The `configureInitialPolicies` method in `gw.job.sxs.PersonalAutoSideBySideInitialConfig` initializes the two new periods. You can modify the code in this class.

The `configureInitialPolicies` method checks to see if the initial period has an offering. If the initial period has an offering, the side-by-side periods are the same as the initial period. If there is no offering, the method applies the basic, standard, and premium offerings to each side-by-side version, respectively.

In policy periods that have the **Standard Program** offering, the `configureInitialPolicies` method also removes `PACollisionCov` and `PAComprehensiveCov` on vehicles over 10 years old.

## Copying Base Data for Side-by-side Quoting

Base data is data which appears to be common to all versions in a side-by-side job (policy transaction). When you make a change to base data, PolicyCenter copies that change to the other side-by-side versions. When PolicyCenter copies the change to the other versions, it creates, removes, and updates entities across all versions as necessary.

When PolicyCenter copies base data to the other side-by-side periods, PolicyCenter opens the target periods for edit if the target periods are quoted. PolicyCenter moves the target periods from quoted to draft status and invalidates the quote.

In the personal auto line of business, some examples of base data are:

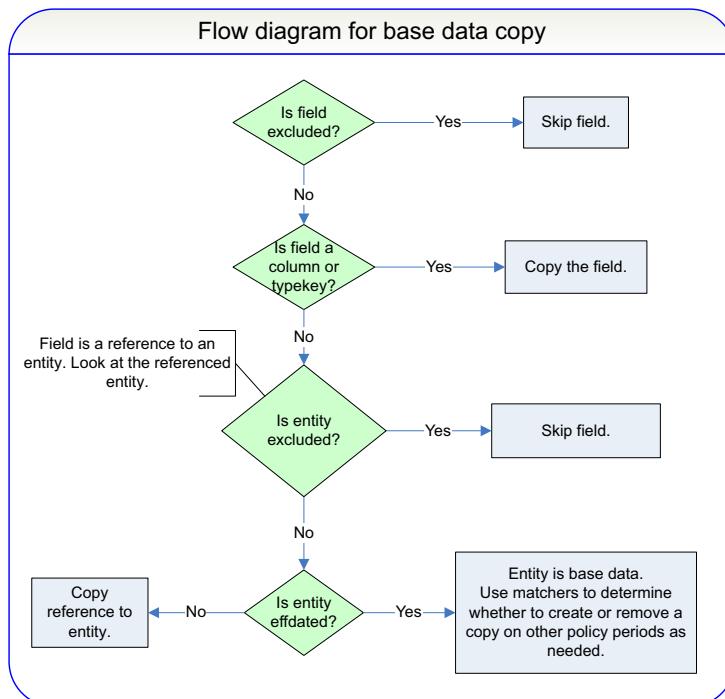
- Policy location
- Driver information such as motor vehicle records
- Line modifiers such as a multi-policy discount

### High-level View of Base Data Copy

Base data copy uses the following logic to traverse the policy period graph to:

- Exclude side-by-side entities and fields from base data copy
- Copy base data entities and fields

The following illustration is a flow diagram for base data copy. Base data copy uses this logic while traversing each node of the policy graph.



If the field is an array, one-to-one, foreign key, or edge foreign key, then base data copy checks to see if the entity pointed to is effective dated. Revised entities implement the `EffDated` delegate. Base data copy skips arrays of entities that are not effective dated because they are implicitly side-by-side data. Base data copy stops recursion on an entity that is side-by-side.

For a complete list of excluded entities and fields, see “[Excluding Side-by-side Data from Base Data](#)” on page 647.

## Gosu Methods for Base Data Copy

Base data copy is implemented by the `SideBySideBaseDataCopy` method in `gw.job.sxs`. When base data is modified on an underlying `PolicyPeriod`, this method copies the change to the other side-by-side policy periods. As necessary, this method creates and removes entities so that the base data appears to be shared among the versions in the side-by-side job.

**Note:** In the policy graph, data that is not base data is referred to as side-by-side data. Unlike base data, side-by-side data is not synchronized between the policy graphs.

The `SideBySideBaseDataCopy` class uses the `MatchableTreeTraverser` class to traverse the policy graph of the source and destination periods, collecting information for use during base data copy. Base data copy starts at `PolicyPeriod` then recursively traverses each link until reaching data excluded from base data.

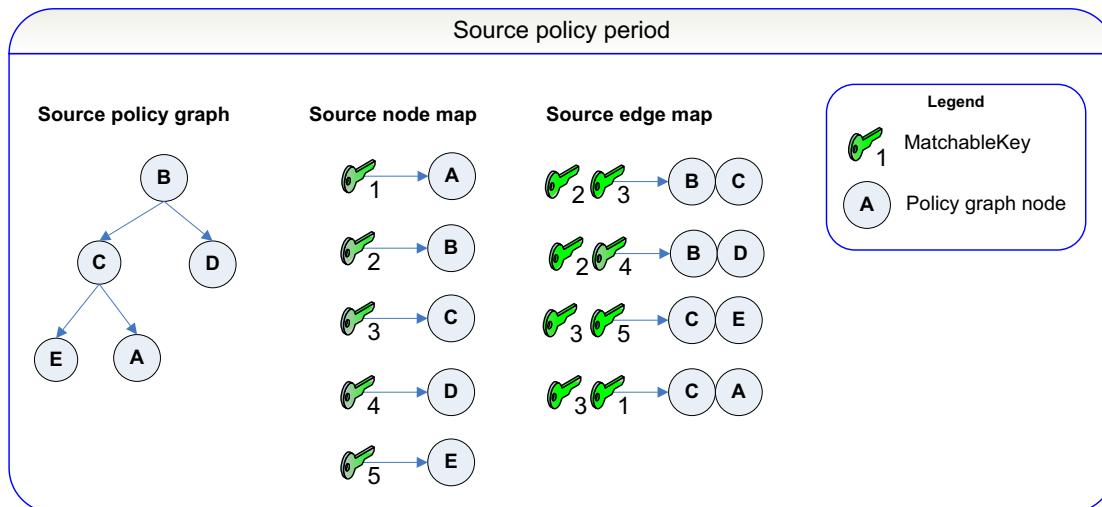
In `SideBySideBaseDataCopy`, the main method for copying base data is the `copySlice` method. This method has three phases.

### Phase 1: Build map of matchable keys and edge map

The `copySlice` method calls the `MatchableTreeTraverser` class to build a map of `MatchableKey` objects for each node in the policy graph. The `MatchableKey` objects are used to determine if nodes from two different policy periods are logically equivalent.

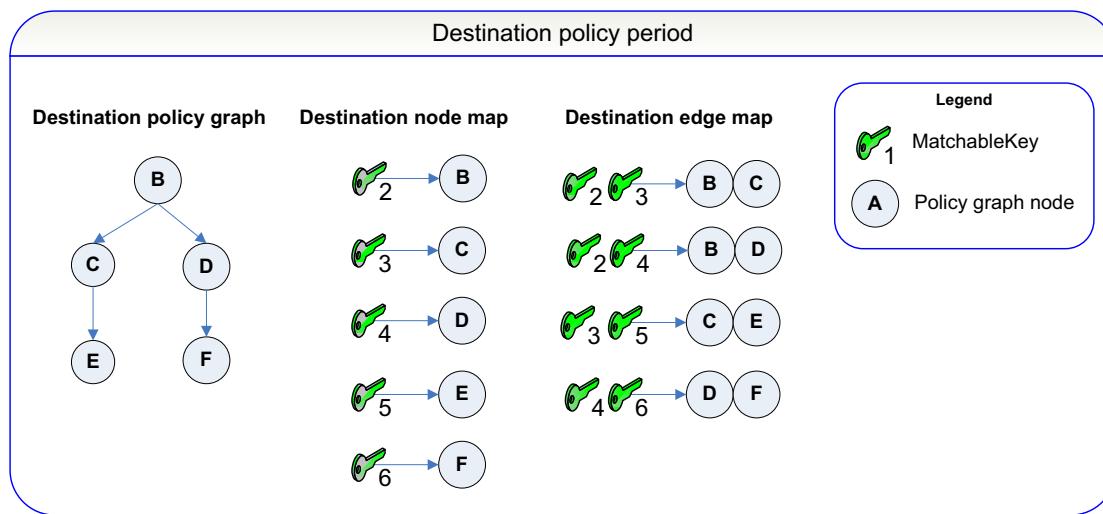
The `copySlice` method also calls the `MatchableTreeTraverser` class to build an edge map. The edge map contains a set of node to node edges for each relationship between nodes in the policy graph.

The following illustration shows the node and edge maps for the policy graph of a source policy period.



The `copySlice` method calls `MatchableTreeTraverser` to build node and edge maps for each destination policy period.

The following illustration shows the node and edge maps for the policy graph of a destination policy period.

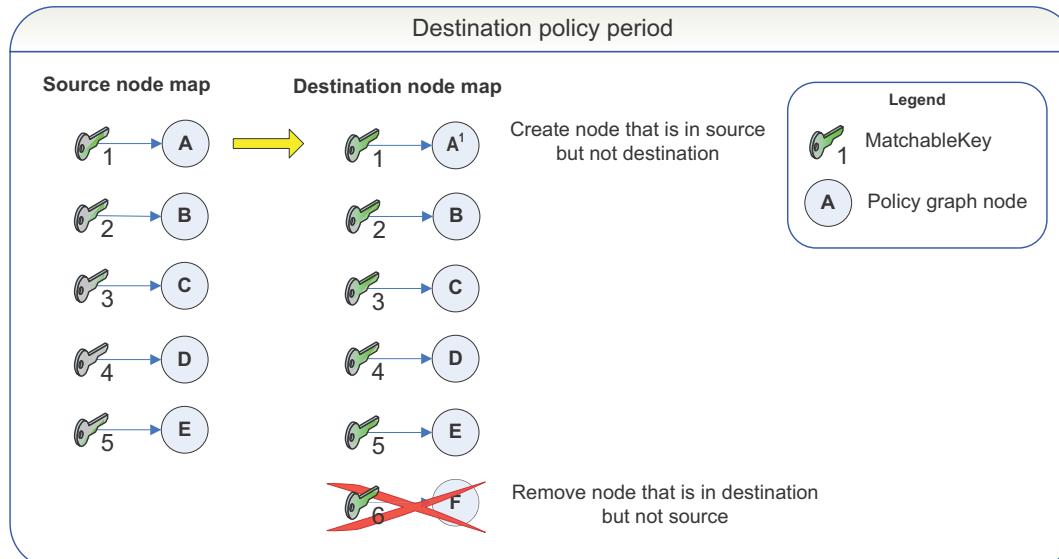


### Phase 2: Compare source and destination node maps

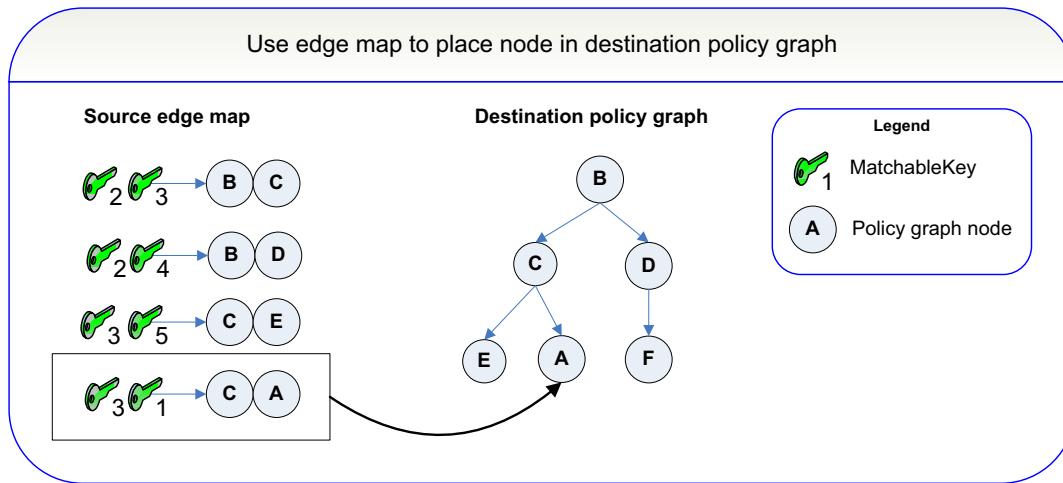
The `copySlice` method compares the source and destination node maps and creates or removes nodes from the destination policy graph.

In the following illustration, MatchableKey 1 exists in the source but not the destination. Therefore, `copySlice` creates a corresponding MatchableKey and a copy of the A node in the destination.

The MatchableKey 6 exists only in the destination node map. Therefore, `copySlice` removes the MatchableKey and the node.



Using the edge map of the source, the new node is inserted into the policy graph of the destination.



### Phase 3: Update field values in destination

The `copySlice` method updates field values on the destination policy graph to match values on the source.

## Excluding Side-by-side Data from Base Data

Side-by-side data are types and fields that PolicyCenter excludes from base data. Types include entities and delegates. Excluding these entities and fields prevents PolicyCenter from copying base data in a way that would cause inconsistencies in the policy data.

In PolicyCenter, the levels of excluded entities and fields are:

- “Side-by-side Data that Base Data Copy Must Exclude” on page 647 – These are not configurable.
- “Side-by-side Data that Base Data Copy Can Exclude” on page 648 – These are configurable.
- “Side-by-side Data in Personal Auto Excluded from Base Data Copy” on page 650 – These are configurable.

**Note:** PolicyCenter also excludes data which is only accessible through excluded entities.

### Side-by-side Data that Base Data Copy Must Exclude

**Note:** You cannot configure the side-by-side data that Guidewire excludes.

Guidewire excludes the following entities and fields from base data copy:

Exclude from base data	Details
Non-effective dated entities	Exclude all entities that are not EffDated.
Autonumber fields	Exclude all fields that point to AutoNumberSequence.
Costs and transactions	Exclude entities and their subtypes that implement the following delegates because the rating engine maintains them: <ul style="list-style-type: none"> <li>Cost</li> <li>Transaction</li> </ul>

Exclude from base data	Details
Policy period	Exclude the following fields on the PolicyPeriod entity and its subtypes: <ul style="list-style-type: none"><li>• ActiveWorkflow</li><li>• BranchName</li><li>• BranchNumber</li><li>• FailedDOSEEvaluation</li><li>• FailedDOSEValidation</li><li>• Job</li><li>• Policy</li><li>• PolicyTerm</li><li>• PrimaryInsuredName (denorm)</li><li>• Status</li><li>• ValidQuote</li><li>• Workflows</li></ul>
Archivable	Exclude all fields on entities that implement the Archivable delegate.
Policy location	Exclude the following field on PolicyLocation: <ul style="list-style-type: none"><li>• LocationNum</li></ul>
Building	Exclude the following field on Building: <ul style="list-style-type: none"><li>• BuildingNum</li></ul>
Policy contact role	Exclude the following fields on PolicyContactRole: <ul style="list-style-type: none"><li>• ContactDenorm</li><li>• SeqNumber</li></ul>

## Side-by-side Data that Base Data Copy Can Exclude

The data excluded from base data copy on all lines of business are defined in Gosu and can be configured.

The following table lists data on all lines of business that base data copy excludes in the default configuration. Guidewire strongly suggests that your implementation also exclude these.

Exclude from base data	Details
Coverages Exclusions Conditions	Exclude entities and their subtypes that implement the following delegates because the Product Model and other availability logic determine whether to add or remove them: <ul style="list-style-type: none"><li>• Coverage</li><li>• Exclusion</li><li>• PolicyCondition</li></ul> Exclude the following entities and their subtypes: <ul style="list-style-type: none"><li>• CoverageSymbol</li><li>• CoverageSymbolGroup</li></ul> Exclude the following fields on entities that implement the Coverable delegate: <ul style="list-style-type: none"><li>• InitialCoveragesCreated</li><li>• InitialExclusionsCreated</li><li>• InitialConditionsCreated</li></ul>
Forms	Forms inference determines whether to add or remove forms based on the contents of the policy. Therefore, exclude the following entities and their subtypes: <ul style="list-style-type: none"><li>• Form</li><li>• FormAssociation</li><li>• FormEdgeTable</li></ul>
Reinsurance	Reinsurance Management determines whether to add or remove reinsurance-related entities and fields based on the contents of the policy. Therefore, exclude the following entity and its subtypes: <ul style="list-style-type: none"><li>• Reinsurable</li></ul> Exclude the following field on PolicyPeriod: <ul style="list-style-type: none"><li>• RIRiskVersionList</li></ul>

Exclude from base data	Details
Underwriting Issues	<p>Underwriting issues have complex logic that determines whether to add or remove issues. Underwriting issues also have an interaction with Policy. Therefore, exclude the following entity and its subtypes:</p> <ul style="list-style-type: none"> <li>UWIssue</li> </ul> <p>The following fields are excluded on PolicyPeriod:</p> <ul style="list-style-type: none"> <li>QuoteHidden</li> <li>EditLocked</li> </ul>
Grandfathering	<p>Exclude fields related to grandfathering on an entity that implements the following delegates:</p> <ul style="list-style-type: none"> <li>Coverage</li> <li>Exclusion</li> <li>Modifier</li> </ul> <p>In the default configuration, the following field is related to grandfathering, and therefore excluded:</p> <ul style="list-style-type: none"> <li>ReferenceDateInternal</li> </ul> <p>Some entities, such as PersonalAutoLine, implement the ReferenceDateInternal field directly rather than through a delegate. Exclude these fields.</p>
Billing	<p>Exclude billing and payment-related fields because they are managed indirectly. Therefore, exclude the following fields on PolicyPeriod:</p> <ul style="list-style-type: none"> <li>DepositCollected</li> <li>InvoiceStreamCode</li> <li>NewInvoiceStream</li> <li>ReportingPatternCode</li> <li>SingleCheckingPatternCode</li> <li>SeriesCheckingPatternCode</li> <li>OverrideBillingAllocation</li> <li>BillImmediatelyPercentage</li> <li>DepositOverridePct</li> <li>DepositAmount</li> <li>WaiveDepositChange</li> <li>PaymentPlanID</li> <li>PaymentPlanName</li> <li>AltBillingAccountNumber</li> <li>AllocationOfRemainder</li> <li>RefuncCalcMethod</li> <li>BillingMethod</li> <li>MinimumPremium</li> <li>PaymentPlans</li> </ul> <p>Exclude the following field on PolicyLine and its subtypes:</p> <ul style="list-style-type: none"> <li>MinumumPremium</li> </ul>
Reporting fields	<p>Exclude the following reporting-related denormalization fields on PolicyPeriod:</p> <ul style="list-style-type: none"> <li>TotalCostRPT</li> <li>TotalPremiumRPT</li> <li>TransactionCostRPT</li> <li>TransactionPremiumRPT</li> </ul>
Additional policy-related fields	<p>Exclude the following fields on PolicyPeriod:</p> <ul style="list-style-type: none"> <li>BaseState</li> <li>WrittenDate</li> </ul> <p>Exclude the following field on PolicyLine:</p> <ul style="list-style-type: none"> <li>NumAddInsured</li> </ul>

## Side-by-side Data in Personal Auto Excluded from Base Data Copy

The following are strongly-suggested excluded data in the personal auto line of business. In the default configuration, PolicyCenter excludes these entities and fields.

Exclude from base data	Details
Grandfathering	Exclude the following field on PersonalAutoLine and PersonalVehicle: <ul style="list-style-type: none"><li>• ReferenceDateInternal</li></ul>
Quick quote	Exclude the following field on PersonalVehicle and PolicyDriver: <ul style="list-style-type: none"><li>• QuickQuoteNumber</li></ul>
Personal auto	Exclude the following entities: <ul style="list-style-type: none"><li>• PersonalVehicle</li><li>• PersonalAutoCov</li></ul> Exclude the following field: <ul style="list-style-type: none"><li>• EffectiveDatedFields.OfferingCode</li></ul>
Two paths to entity	Exclude the following entities because they have pointers to side-by-side entities, and there are potentially two paths to reach them: <ul style="list-style-type: none"><li>• VehicleDriver</li><li>• PAVehicleAddlInterest</li></ul> See "Exclude Entities Reachable Through Two Paths" on page 650.

The side-by-side data are configured in the `gw.job.sxs.PersonalAutoSideBySideBaseDataConfig` class.

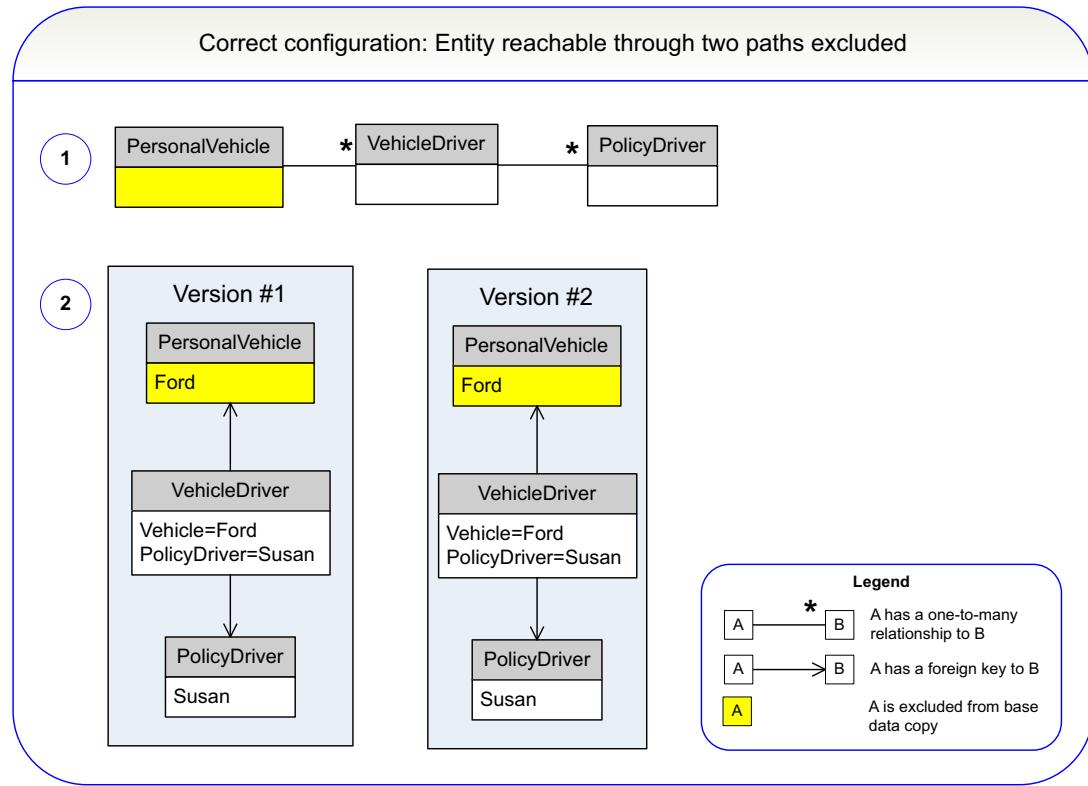
## Exclude Entities Reachable Through Two Paths

Base data copy can produce incorrect results in the following situation:

- If an entity is not excluded, and
- That entity is reachable by foreign key or array through two paths, one of which is through an excluded entity and the other is not.

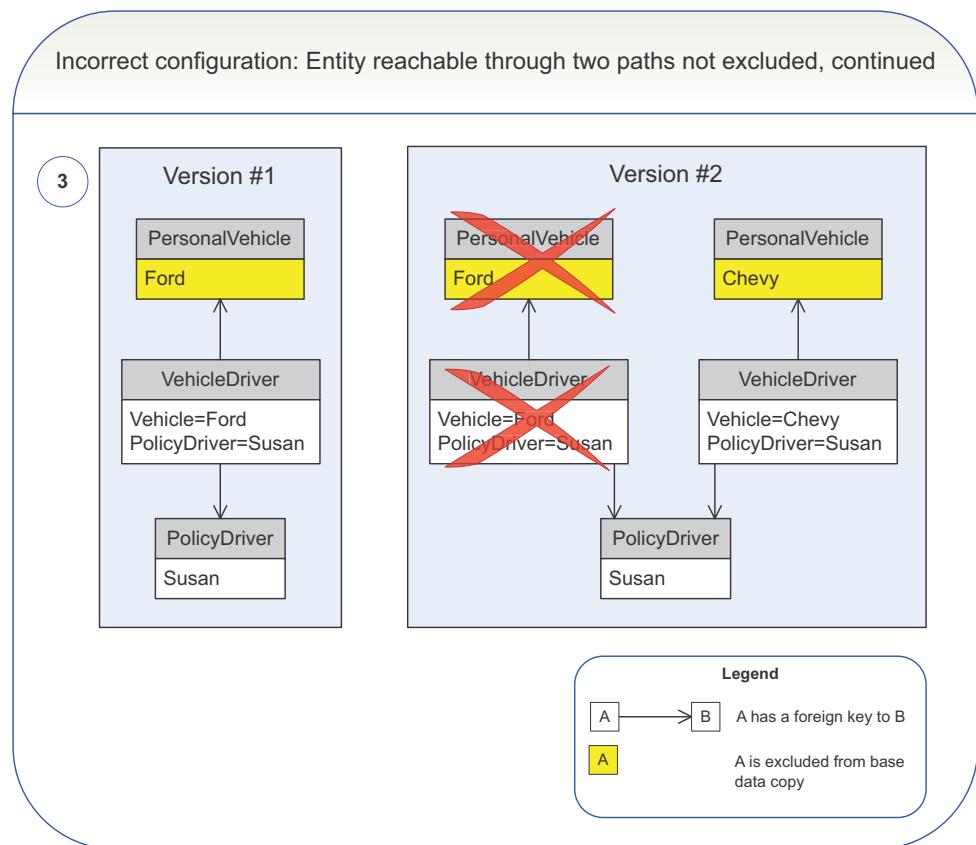
**Note:** Guidewire recommends that you exclude an entity reachable through two paths if one of the paths is through an excluded entity. Guidewire especially recommends that an entity reachable through two paths be excluded if the foreign key is non-nullable.

In personal auto, **VehicleDriver** entity provides an example as shown in the following illustration of an incorrect configuration.



1. The **VehicleDriver** entity can be reached through **PolicyDriver** and **PersonalVehicle**. However, only **PersonalVehicle** is excluded from base data copy.
2. Initially, **PersonalVehicle** and **VehicleDriver** in Version #1 and Version #2 have the same values.

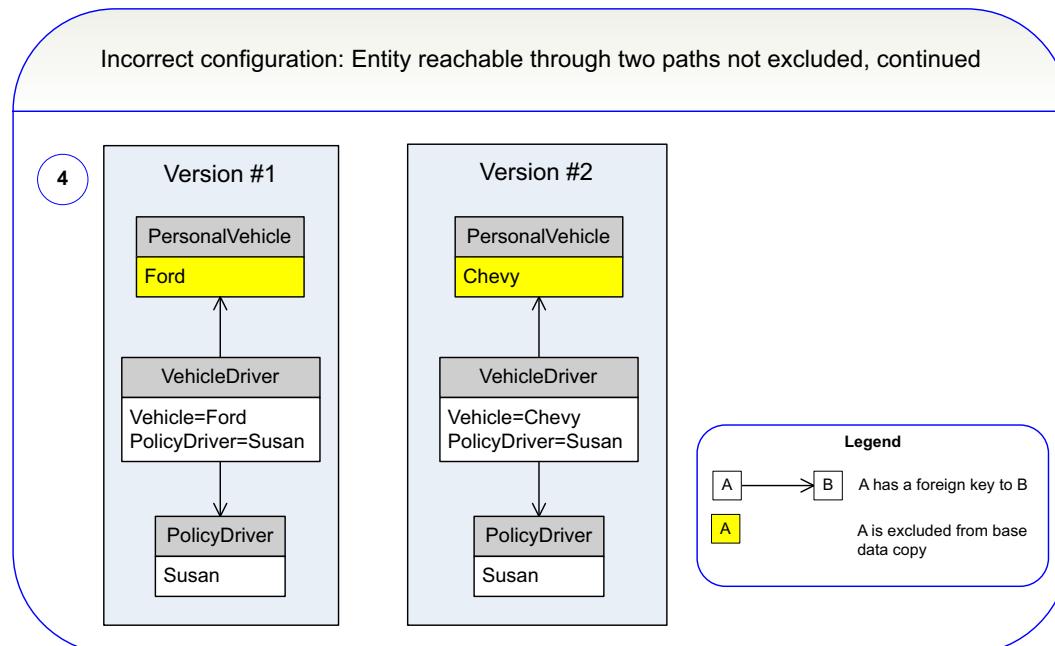
The following illustration shows changes that happen when the agent edits policy Version #2.



3. The agent edits Version #2 and makes the following changes:

- Removes the Ford and the VehicleDriver connecting Susan to the Ford.
- Adds a Chevy and a VehicleDriver connecting Susan to the Chevy.

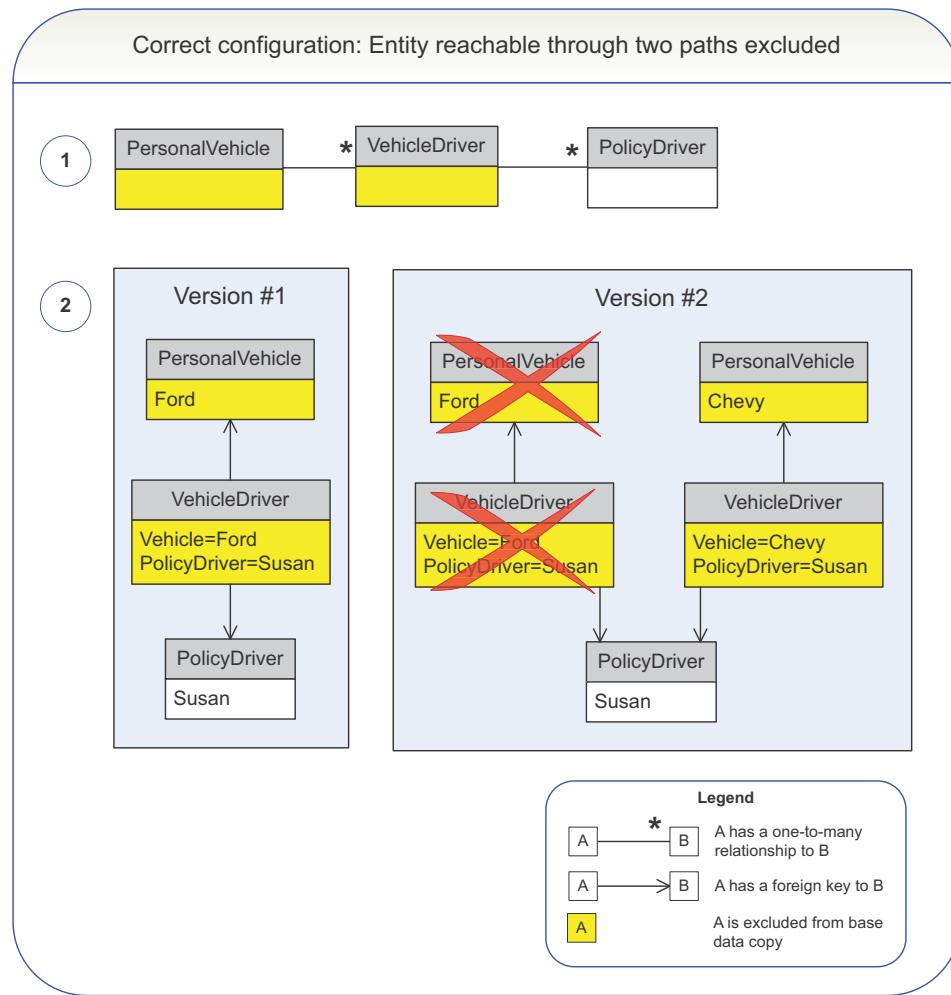
The following illustration shows the problems that base data copy encounters.



#### 4. When the agent saves the edits, PolicyCenter does the base data copy.

Removing the Ford and adding the Chevy is not a problem because `PersonalVehicle` is side-by-side data and does not affect other versions. However, `VehicleDriver` is a base data entity and must be the same in all versions. In Version #1, the `Vehicle` field points to the Ford. In Version #2, it points to the Chevy. Version #1 would contain bad data if base data copies `Vehicle=Chevy` to Version #1.

The `VehicleDriver` entity is reachable through two paths. Therefore, Guidewire recommends making the `VehicleDriver` entity side-by-side data by excluding it from base data copy, as shown in the following illustration.



1. Base data copy excludes both the `PersonalVehicle` and `VehicleDriver` entities.

2. It does not matter that the `PersonalVehicle` and `VehicleDriver` differ between the versions because base data copy excludes them.

## Configuring Data Excluded from Base Data

To make data side-by-side data, you must exclude the field or type from base data. Fields include foreign keys, edge foreign keys, arrays, and one-to-one relationships. Fields also include scalar values such as bit, integer, and text. Types include entities and delegates. To exclude a field or type, add it to the `gw.job.sxs.PersonalAutoSideBySideBaseDataConfig` class:

- Add your new field to `sideBySideProperties`
- Add your new side-by-side type to `sideBySideTypes`

## Configuring Quote All to Ignore Validation Warnings

In the default configuration, **Quote All** in the **Side by Side Quoting** screen produces quotes only if there are no validation warnings. This behavior prevents PolicyCenter from generating invalid quotes. Guidewire expects that the default implementation is suitable for most implementations.

However, in some limited cases, your implementation may need to **Quote All** even though there are validation warnings. To quote when there are validation warnings, set the `SideBySideQuoteBlockedByWarnings` parameter in `config.xml` to `false`. The default value is `true`. If this parameter is not in `config.xml`, you need to add it.

Changing this parameter can affect product configuration files and Gosu code. Some code may assume that side-by-side quoting only occurs if there are no validation warnings. Test thoroughly to avoid problems.

In `gw.job.sxs.SideBySideProcess`, in the call to the `requestQuote` method has the `SideBySide` parameter as an argument.