

Guidewire ClaimCenter®

ClaimCenter Configuration Guide

RELEASE 8.0.2

Guidewire®
Deliver insurance your way™

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 ClaimCenter

Product Release: 8.0.2

Document Name: *ClaimCenter Configuration Guide*

Document Revision: 20-May-2014

Contents

About ClaimCenter Documentation	23
Conventions in This Document	24
Support	24

Part I ClaimCenter Configuration Basics

1 Overview of ClaimCenter Configuration.....	27
What You Can Configure	27
How You Configure ClaimCenter	28
Types of Application Environments	29
The Development Environment	29
The Production Environment	29
Deploying Configuration Files	29
Deploying Changes in a Development Environment	29
Deploying Changes to the Production Server	30
Regenerating the Data Dictionary and Security Dictionary	30
Generating the Data and Security Dictionaries in HTML Format	31
Generating the Data and Security Dictionaries in XML Format	31
Generating the Data and Security Dictionaries as You Generate a .war or .ear File	31
Aspects of Regenerating the Security Dictionary	31
Managing Configuration Changes	32
2 Application Configuration Parameters	33
Working with Configuration Parameters	34
Accessing Configuration Parameters in Gosu	34
Configuration Parameter Attributes	35
Adding Custom MIME Types	35
Approval Parameters	36
BulkInvoiceApprovalPattern	36
PaymentApprovalPattern	36
RecoveryApprovalPattern	36
RecoveryReserveApprovalPattern	36
ReserveApprovalPattern	36
Archive Parameters	36
ArchiveEnabled	37
AssignClaimToRetriever	37
DaysClosedBeforeArchive	37
DaysRetrievedBeforeArchive	38
PolicySystemArchivingEnabled	38
RestorePattern	38
SnapshotEncryptionUpgradeChunkSize	38
Assignment Parameters	38
AssignmentQueuesEnabled	39
WeightedAssignmentEnabled	39
WeightedAssignmentGlobalDefaultWeight	39
Batch Process Parameters	39
BatchProcessHistoryPurgeDaysOld	39

Business Calendar Parameters	39
BusinessDayDemarcation	39
BusinessDayEnd	39
BusinessDayStart	40
BusinessWeekEnd	40
HolidayList (Obsolete)	40
IsFridayBusinessDay	40
IsMondayBusinessDay	40
IsSaturdayBusinessDay	40
IsSundayBusinessDay	40
IsThursdayBusinessDay	40
IsTuesdayBusinessDay	41
IsWednesdayBusinessDay	41
MaxAllowedDate	41
MinAllowedDate	41
Cache Parameters	41
ActivityPatternCacheMaxDuration	41
ExchangeRatesCacheRefreshIntervalSecs	41
GlobalCacheActiveTimeMinutes	42
GlobalCacheDetailedStats	42
GlobalCacheReapingTimeMinutes	42
GlobalCacheSizeMegabytes	42
GlobalCacheSizePercent	42
GlobalCacheStaleTimeMinutes	43
GlobalCacheStatsRetentionPeriodDays	43
GlobalCacheStatsWindowMinutes	43
GroupCacheRefreshIntervalSecs	43
ScriptParametersRefreshIntervalSecs	43
TreeViewRefresh	43
ZoneCacheRefreshIntervalSecs	44
Claim Catastrophe Parameters	44
HeatMapCredential	44
HeatMapServiceTemplate	44
MaxCatastropheClaimFinderSearchResults	44
Claim Health Indicator and Metric Parameters	44
ClaimHealthCalcMaxLossDateInYears	44
InitialReserveAllowedPeriod	45
MaxClaimResultsPerClaimHealthCalcBatch	45
Clustering Parameters	45
ClusteringEnabled	45
ClusterMemberPurgeDaysOld	46
ClusterMemberRecordUpdateIntervalSecs	46
ClusterMulticastAddress	46
ClusterMulticastPort	46
ClusterMulticastTTL	46
ClusterProtocolStackOption1	46
ClusterProtocolStackOption2	47
ClusterProtocolStack	47
ClusterStatisticsMonitorIntervalMins	47
ConfigVerificationEnabled	47
PDFMergeHandlerLicenseKey	48

Database Parameters.....	48
DisableHashJoinForClaimSearch	48
DisableHashJoinForTeamGroupActivities	48
DisableIndexFastFullScanForClaimSearch	48
DisableIndexFastFullScanForRecoverySearch.....	48
DisableIndexFastFullScanForTeamGroupActivities	48
DisableSortMergeJoinForClaimSearch.....	49
DisableSortMergeJoinForTeamGroupActivities	49
DiscardQueryPlansDuringStatsUpdateBatch	49
IdentifyQueriesViaComments.....	49
MigrateToLargeIDsAndDatetime2.....	49
QueryRewriteForClaimSearch	49
SetSemiJoinNestedLoopsForClaimSearch	51
Deduction Parameters.....	51
BackupWithholdingTypeCode	51
CalculateBackupWithholdingDeduction.....	51
StandardWithholdingRate	51
Document Creation and Document Management Parameters	52
AllowDocumentAssistant	52
DisplayDocumentEditUploadButtons.....	52
DocumentAssistantJNLP	52
DocumentContentDispositionMode	52
DocumentTemplateDescriptorXSDLocation	52
MaximumFileUploadSize	52
UseDocumentAssistantToDisplayDocuments	53
Domain Graph Parameters	53
DomainGraphKnownLinksWithIssues.....	53
DomainGraphKnownUnreachableTables.....	53
Environment Parameters.....	53
AddressVerificationFailureAsError	53
AlwaysShowPhoneWidgetRegionCode	54
CurrentEncryptionPlugin	54
DeprecatedEventGeneration	54
EnableAddressVerification	54
EnableInternalDebugTools	55
EntityValidationOrder	55
KeyGeneratorRangeSize	55
MemoryUsageMonitorIntervalMins	55
PublicIDPrefix	55
ResourcesMutable	56
RetainDebugInfo	56
StrictDataTypes	56
TwoDigitYearThreshold	57
UnreachableCodeDetection	57
UnrestrictedUserName	57
UseOldStylePreUpdate	57
WarnOnImplicitCoercion	58
WebResourcesDir	58

Financial Parameters.....	58
AllowMultipleLineItems.....	58
AllowMultiplePayments	58
AllowNoPriorPaymentSupplement.....	58
AllowPaymentsExceedReservesLimits.....	58
CheckAuthorityLimits	59
CloseClaimAfterFinalPayment	59
CloseExposureAfterFinalPayment	59
EnableMulticurrencyReserving.....	59
ExchangeRatesCacheRefreshIntervalSecs	59
Financials	59
PaymentLogThreshold	59
PaymentRoundingMode	60
ReserveRoundingMode.....	60
SetReservesByTotalIncurred.....	61
UseDeductibleHandling	61
UseRecoveryReserves.....	61
Geocoding Feature Parameters.....	61
UseGeocodingInPrimaryApp	61
UseGeocodingInAddressBook	61
ProximitySearchOrdinalMaxDistance	62
ProximityRadiusSearchDefaultMaxResultCount	62
UseMetricDistancesByDefault	62
Globalization Parameters	62
DefaultApplicationLanguage	63
DefaultApplicationLocale	63
DefaultApplicationCurrency.....	63
DefaultRoundingMode	63
MulticurrencyDisplayMode	64
DefaultCountryCode	64
DefaultPhoneCountryCode	64
DefaultNANPACountryCode	65
AlwaysShowPhoneWidgetRegionCode	65
Integration Parameters	65
ContactAutoSyncWorkItemChunkSize	65
DefaultXmlExportEncryptionId	65
EnableMetroIntegration	65
InstantaneousContactAutoSync	66
ISOPropertiesFileName	66
KeepCompletedMessagesForDays	66
LoadSoapServicesOnStartup.....	66
LockPrimaryEntityDuringMessageHandling	66
MetroPropertiesFileName	67
PluginStartupTimeout	67
PolicySystemURL.....	67
UseSafeBundleForWebServicesOperations	67
Miscellaneous Bulk Invoice Activity Pattern Parameters	68
BulkInvoiceItemValidationFailedPattern	68
BulkInvoiceUnableToStopPattern	68
BulkInvoiceUnableToVoidPattern	68

Miscellaneous Financial Activity Parameters.....	68
CheckDeniedPattern	69
CheckUnableToStopPattern	69
CheckUnableToVoidPattern	69
LastPaymentReminderPattern.....	69
RecoveryDeniedPattern.....	69
Miscellaneous Parameters.....	69
ClaimLossDateDemarcation.....	69
ConsistencyCheckerThreads.....	69
EnableClaimNumberGeneration.....	69
EnableClaimSnapshot.....	70
EnableStatCoding	70
JGroupsClusterChannel	70
ListViewPageSizeDefault	70
MaintainPolicyValidationLevelOnPolicyChange.....	70
MaxCachedClaimSnapshots	70
MaxStatCodesInDropdown.....	70
ProfilerDataPurgeDaysOld	70
VendorNotificationAPIRetryTime	71
TransactionIdPurgeDaysOld.....	71
PDF Print Settings Parameters	71
DefaultContentDispositionMode	71
PrintFontFamilyName.....	71
PrintFontSize.....	71
PrintFOPUserConfigFile.....	71
PrintHeaderFontSize	72
PrintLineHeight.....	72
PrintListViewBlockSize	72
PrintListViewFontSize	72
PrintMarginBottom.....	72
PrintMarginLeft	72
PrintMarginRight	72
PrintMarginTop.....	72
PrintMaxPDFInputFileSize.....	73
PrintPageHeight	73
PrintPageWidth	73
Scheduler and Workflow Parameters	73
IdleClaimThresholdDays	73
SchedulerEnabled	73
SeparateIdleClaimExceptionMonitor	74
WorkflowLogDebug	74
WorkflowLogPurgeDaysOld	74
WorkflowPurgeDaysOld.....	74
WorkflowStatsIntervalMins	74

Search Parameters	74
FreeTextSearchEnabled	74
MaxActivitySearchResults	75
MaxBulkInvoiceSearchResults	75
MaxCheckSearchResults	75
MaxClaimSearchResults	75
MaxContactSearchResults	75
MaxDocTemplateSearchResults	75
MaxDuplicateContactSearchResults	75
MaxNoteSearchResults	76
MaxPolicySearchResults	76
MaxRecoverySearchResults	76
SetSemiJoinNestedLoopsForClaimSearch	76
Security Parameters	76
EnableDownlinePermissions	76
FailedAttemptsBeforeLockout	76
LockoutPeriod	77
LoginRetryDelay	77
MaxACLParameters	77
MaxPasswordLength	78
MinPasswordLength	78
RestrictContactPotentialMatchToPermittedItems	78
RestrictSearchesToPermittedItems	78
SessionTimeoutSecs	78
ShouldSyncUserRolesInLDAP	78
UseACLPermissions	78
Segmentation Parameters	79
ClaimSegment	79
ClaimStrategy	79
ExposureSegment	79
ExposureStrategy	79
Service Request Parameters	79
ServiceRequestAPIMaxDaysKeepActiveWithoutInvoice	79
ServiceRequestAPIMaxMessageResults	79
ServiceRequestAPIMaxSearchResults	80
Spell Check Parameters	80
CheckSpellingOnChange	80
CheckSpellingOnDemand	80
Statistics, Team, and Dashboard Parameters	80
AgingStatsFirstDivision	80
AgingStatsSecondDivision	80
AgingStatsThirdDivision	81
CalculateLitigatedClaimAgingStats	81
DashboardIncurredLimit	81
DashboardShowByCoverage	81
DashboardShowByLineOrLoss	81
DashboardWindowPeriod	81
GroupSummaryShowUserGlobalWorkloadStats	81
UserStatisticsWindowSize	81

User Interface Parameters.....	82
ActionsShortcut.....	82
AutoCompleteLimit	82
EnableClaimantCoverageUniquenessConstraint	82
HidePolicyObjectsWithNoCov...ragesForLossTypes	82
HighlyLinkedContactThreshold	82
IgnorePolicyTotalPropertiesValue	82
IgnorePolicyTotalVehiclesValue	83
InputHelpTextOnFocus.....	83
InputHelpTextOnMouseOver	83
InputMaskPlaceholderCharacter	83
ListViewPageSizeDefault	83
MaxBrowserHistoryItems (Obsolete).....	83
MaxChooseByCoverageMenuItems	83
MaxChooseByCoverageTypeMenuItems.....	84
MaxClaimantsInClaimListViews	84
MaxTeamSummaryChartUserBars.....	84
QuickJumpShortcut.....	84
RequestReopenExplanationForTypes.....	84
ShowCurrentPolicyNumberInSelectPolicyDialog	84
ShowFixedExposuresInLossDetails	84
ShowNewExposureChooseByCoverageMenuForLossTypes.....	84
ShowNewExposureChooseByCoverageTypeMenuForLossTypes.....	85
ShowNewExposureMenuForLossTypes.....	85
UISkin	85
WizardNextShortcut	85
WizardPrevShortcut	85
WizardPrevNextButtonsVisible	85
Work Queue Parameters.....	85
InstrumentedWorkerInfoPurgeDaysOld.....	86
WorkItemCreationBatchSize	86
WorkItemRetryLimit	86
WorkQueueHistoryMaxDownload	86
WorkQueueThreadPoolMaxSize	86
WorkQueueThreadPoolMinSize	86
WorkQueueThreadsKeepAliveTime.....	86

Part II

The Guidewire Development Environment

3 Working with Guidewire Studio.....	89
What Is Guidewire Studio?.....	89
Starting Guidewire Studio	90
Restarting Studio.....	90
The Studio Development Environment	90
Working with the QuickStart Development Server	91
Connecting the Development Server to a Database	92
Deploying Your Configuration Changes	93
ClaimCenter Configuration Files	93
Key Directories	94
4 ClaimCenter Studio and Gosu.....	97
Studio and the DCE VM.....	97
Gosu Building Blocks.....	98

Gosu Case Sensitivity	99
Working with Gosu in ClaimCenter Studio	100
Gosu Packages	100
Gosu Classes	100
ClaimCenter Base Configuration Classes	101
Class Visibility in Studio	103
Preloading Gosu Classes	103
Gosu Enhancements	104
The Guidewire XML Model	105
Script Parameters	105
Script Parameters Overview	105
Working with Script Parameters	106
Referencing a Script Parameter in Gosu	107

Part III

Guidewire Studio Editors

5 Using the Studio Editors	111
Editing in Guidewire Studio	111
Working in the Gosu Editor	112
6 Using the Plugins Registry Editor	113
What Are Plugins?	113
Plugin Implementation Classes	113
What is the Plugins Registry?	113
Startable Plugins	114
Working with Plugins	114
Creating a New Plugins Registry Item	114
Adding a New Plugin Interface Implementation	114
Setting Environment and Server Context for Plugin Implementations	115
Customizing Plugin Functionality	116
Working with Plugin Versions	116
7 Working with Web Services	119
Web Services and Guidewire Studio	119
Using the RPC-Encoded Web Services Editor	120
Using the WS-I Web Service Editor	122
Defining a Web Service Collection	123
8 Implementing QuickJump Commands	125
What Is QuickJump?	125
Adding a QuickJump Navigation Command	126
Implementing QuickJumpCommandRef Commands	126
Implementing StaticNavigationCommandRef Commands	128
Implementing ContextualNavigationCommandRef Commands	128
Checking Permissions on QuickJump Navigation Commands	128
9 Using the Entity Names Editor	131
Entity Names Editor	131
Variable Table	132
The Entity Path Column	132
The Use Entity Names? Column	132
The Sort Columns	133
Gosu Text Editor	133
Including Data from Subentities	134
Entity Name Types	135

10 Using the Messaging Editor.....	137
Messaging Editor	137
Adding a Messaging Environment	137
Adding a Message Destination	138
Associating Event Names with a Message Destination	141
11 Using the Display Keys Editor.....	143
Display Keys Editor	143
Creating Display Keys in a Gosu Editor.....	144
Retrieving the Value of a Display Key.....	144

Part IV

Data Model Configuration

12 Working with the Data Dictionary	149
What is the Data Dictionary?	149
What Can You View in the Data Dictionary?	150
Using the Data Dictionary	150
Field Colors.....	151
Object Attributes.....	151
Entity Subtypes.....	152
Data Column and Field Types	152
Virtual Properties on Data Entities	153
13 The ClaimCenter Data Model	155
What is the Data Model?	155
The Data Model in Guidewire Application Architecture	156
The Base Data Model	156
Working with Dot Notation	156
Overview of Data Entities.....	157
Data Entity Metadata Files	157
Working with Data Entity Definition Files.....	160
Search for an Existing Entity Definition.....	160
Create a New Entity Definition	160
Extend an Existing Entity Definition	160
ClaimCenter Data Entities	161
Data Entities and the Application Database	161
ClaimCenter Database Tables.....	163
Data Objects and Scriptability	164
Base ClaimCenter Data Objects	166
Component Data Objects	166
Delegate Data Objects.....	167
Delete Entity Data Objects	170
Entity Data Objects	170
Extension Data Objects.....	175
NonPersistent Entity Data Objects	176
Subtype Data Objects	178
viewEntity Data Objects	180
viewEntityExtension Data Objects	182

Data Object Subelements	183
<array>	185
<column>	187
<componentref>	192
<edgeForeignKey>	193
<events>	196
<foreignkey>	197
<fulldescription>	200
<implementsEntity>	200
<implementsInterface>	201
<index>	202
<onetoone>	203
<remove-index>	204
<typekey>	205
14 Modifying the Base Data Model.....	209
Planning Changes to the Base Data Model.....	209
Overview of Data Model Extension	209
Strategies for Extending the Base Data Model	210
What Happens If You Change the Data Model?	211
Naming Restrictions for Extensions	212
Defining a New Data Entity	212
Extending a Base Configuration Entity	213
Working with Attribute Overrides	213
Extending the Base Data Model: Examples	216
Creating a New Delegate Object	216
Extending a Delegate Object	218
Defining a Subtype	221
Defining a Reference Entity	222
Defining an Entity Array	222
Implementing a Many-to-Many Relationship Between Entity Types	223
Extending an Existing View Entity	224
Removing Objects from the Base Configuration Data Model	225
Removing a Base Extension Entity	226
Removing an Extension to a Base Object	227
Implications of Modifying the Data Model	227
Deploying Data Model Changes to the Application Server	229
15 Working with Associative Arrays.....	231
Overview of Associative Arrays	231
Associative Array Mapping Types	232
Scriptability and Associative Arrays	232
Issues with Setting Array Member Values	233
Subtype Mapping Associative Arrays	233
Working with Array Values Using Subtype Mapping	234
Typelist Mapping Associative Arrays	235
Working with Array Values Using Typelist Mapping	236
16 The Domain Graph	239
Domain Graph Overview	239
Object Ownership and the Domain Graph	240
Ownership Through Foreign Keys	240
Reverse Ownership in the Domain Graph	240
Accessing the Domain Graph	241
Viewing the Domain Graph	241

Adding Objects to the Domain Graph	242
Implementing the Correct Delegate	243
Defining Foreign Keys Between Objects	245
Graph Validation Tests	246
Working with Changes to the Data Model	247
Working with Shared Entity Data	247
Working with Cycles	248
17 Field Validation	251
Field Validators	251
Field Validator Definitions	252
<FieldValidators>	253
<ValidatorDef>	253
Modifying Field Validators	255
Using <columnOverride> to Modify Field Validation	255
18 Data Types	257
Overview of Data Types	257
Working with Data Types	258
Using Data Types	258
Defining a Data Type for a Property	259
The Data Types Configuration File	260
<...DataType>	260
Deploying Modifications to Data Types Configuration File	260
Customizing Base Configuration Data Types	261
List of Customizable Data Types	262
Working with the Medium Text Data Type (Oracle)	263
The Data Type API	263
Retrieving the Data Type for a Property	263
Retrieving a Particular Data Type in Gosu	264
Retrieving a Data Type Reflectively	264
Using the IDataType Methods	264
Defining a New Data Type: Required Steps	265
Defining a New Tax Identification Number Data Type	265
Step 1: Register the Data Type	266
Step 2: Implement the IDataTypeDef Interface	266
Step 3: Implement the Data Type Aspect Handlers	267
19 Working with Typelists	271
What is a Typelist?	272
Terms Related to Typelists	272
Typelists and Typecodes	272
Typelist Definition Files	273
Different Kinds of Typelists	274
Internal Typelists	274
Extendable Typelists	275
Custom Typelists	275
Working with Typelists in Studio	275
The Typelists Editor	275
Entering Typecodes	277
Typekey Fields	278
Removing or Retiring a Typekey	280
Removing a Typekey	281
Typelist Filters	282

Static Filters	282
Creating a Static Filter Using Categories	283
Creating a Static Filter Using Includes	285
Creating a Static Filter Using Excludes	286
Dynamic Filters.....	287
Creating a Dynamic Filter.....	288
Typecode References in Gosu	290
Mapping Typecodes to External System Codes	291

Part V User Interface Configuration

20 Using the PCF Editor	295
Page Configuration (PCF) Editor	295
Page Canvas Overview	296
Creating a New PCF File	296
Working with Shared or Included Files	297
Understanding PCF Modes	298
Setting a PCF Mode	298
Creating New Modal PCF files	299
Page Config Menu	299
Toolbox Tab	300
Structure Tab	300
Properties Tab	301
Child Lists	302
PCF Elements	303
PCF Elements and the Properties Tab	303
Working with Elements	303
Adding an Element to the Canvas	304
Moving an Element on the Canvas	304
Changing the Type of an Element	305
Adding a Comment to an Element	305
Finding an Element on the Canvas	306
Viewing the Source of an Element	306
Duplicating an Element	306
Deleting an Element	306
Copying an Element	307
Cutting an Element	307
Pasting an Element	307
Linking Widgets	307
21 Introduction to Page Configuration.....	309
Page Configuration Files	309
Page Configuration Elements	309
What is a PCF Element?	310
Types of PCF Elements	310
Identifying PCF Elements in the User Interface	312
Getting Started Configuring Pages	314
Finding an Existing Element To Edit	314
Creating a New Standalone PCF Element	315

Modifying Style and Theme Elements	316
Changing or Adding Images	316
Overriding CSS	316
Changing Theme Colors	317
Advanced Re-Theming	317
22 Data Panels	319
Panel Overview	319
Detail View Panel	319
Define a Detail View	320
Add Columns to a Detail View	321
Format a Detail View	322
List View Panel	324
Define a List View	325
Iterate a List View Over a Data Set	327
Choose the Data Source for a List View	328
23 Location Groups	331
Location Group Overview	331
Define a Location Group	332
Location Groups as Navigation	333
Location Groups as Tab Menus	333
Location Groups as Menu Links	334
Location Groups as Screen Tabs	334
24 Navigation	337
Navigation Overview	337
Tab Bars	338
Configure the Default Tab Bar	338
Specify Which Tab Bar to Display	338
Define a Tab Bar	339
Tabs	339
Define a Tab	339
Define a Drop-down Menu on a Tab	339
25 Configuring Spell Check	341
Using the Spell Checking Feature	341
Understanding How Spell Checking Works	341
How to Configure Spell Check	342
Step 1: Implement a Spell Check Utility	342
Step 2: Implement a checkSpelling Method	342
Step 3: Set Spelling Checker Parameters in config.xml	343
Step 4: Configure Text Fields for Spell Checking	343
26 Configuring Search Functionality	345
Search Overview	345
Database Search Configuration	347
ClaimCenter Database Search Functionality	347
Configuring ClaimCenter Database Search	348
Deploying Customized Database Search Files	355
Steps in Customizing a Database Search	355

Free-text Search Configuration.....	360
Overview of Free-text Search	360
Free-text Search System Architecture.....	360
Enabling Free-text Search in ClaimCenter	363
Configuring the Guidewire Solr Extension for Integration with ClaimCenter.....	364
Configuring Free-text Search for Indexing and Searching.....	367
Configuring the Free-text Batch Load Command.....	368
Configuring the Search by Contact Screen for Free-text Search	368
Modifying Free-text Search for Additional Fields	369
27 Configuring Special Page Functions	373
Adding Print Capabilities	373
Overview of the Print Functionality	373
Types of Printing Configuration	375
Working with a PrintOut Page	376
Overriding the Print Settings in a File	379
Troubleshooting Print Configurations	379
Linking to a Specific Page: Using an EntryPoint PCF.....	379
Entry Points.....	380
Creating a Forwarding EntryPoint PCF	381
Linking to a Specific Page: Using an ExitPoint PCF	382
Creating an ExitPoint PCF	382

Part VI

Workflow and Activity Configuration

28 Using the Workflow Editor.....	387
Workflow in Guidewire ClaimCenter.....	387
Workflow in Guidewire Studio.....	387
Understanding Workflow Steps	389
Using the Workflow Right-Click Menu	390
Using Search with Workflow	390
29 Guidewire Workflow	393
Understanding Workflow	394
Workflow Instances	394
Work Items	395
Workflow Process Format.....	395
Workflow Step Summary	396
Workflow Gosu.....	396
Workflow Versioning	397
Workflow Localization	398
Workflow Structural Elements	398
<Context>	399
<Start>.....	399
<Finish>	399
Common Step Elements	399
Enter and Exit Scripts	400
Asserts.....	400
Events	400
Notifications	401
Branch IDs	401

Basic Workflow Steps	401
AutoStep	401
MessageStep	402
ActivityStep	403
ManualStep	404
Outcome	405
Step Branches	406
Working with Branch IDs.....	407
GO.....	407
TRIGGER.....	408
TIMEOUT.....	409
Creating New Workflows.....	411
Cloning an Existing Workflow.....	411
Extending an Existing Workflow	411
Extending a Workflow: A Simple Example	412
Instantiating a Workflow	414
A Simple Example of Instantiation.....	415
The Workflow Engine	417
Distributed Execution	417
Synchronicity, Transactions, and Errors.....	417
Workflow Subflows	420
Workflow Administration.....	421
Workflow Debugging and Logging	423
30 Defining Activity Patterns	425
What is an Activity Pattern?.....	425
Pattern Types and Categories	426
Activity Pattern Types	426
Categorizing Activity Patterns	427
Using Activity Patterns in Gosu	427
Calculating Activity Due Dates	427
Target Due Dates (Deadlines).....	428
Escalation Dates	428
Defining the Business Calendar	428
Configuring Activity Patterns.....	429
Using Activity Patterns with Documents and Emails.....	431
Localizing Activity Patterns	431
Part VII	
Testing Gosu Code	
31 Debugging and Testing Your Gosu Code.....	435
The Studio Debugger	435
Initializing the Application Server for Debugging	436
Starting the Studio Debugger	436
Setting Breakpoints.....	436
Stepping Through Code	437
Viewing Current Values.....	437
Defining a Watch List.....	438
Resuming Execution.....	438
Using the Gosu Scratchpad.....	438
Testing a Gosu Expression	439
Suggestions for Testing Rules	439

32 Using GUnit	441
The TestBase Class	441
Overriding TestBase Methods	442
Configuring the Server Environment	442
Configuring the Test Environment	444
Configuration Parameters	444
Creating a GUnit Test Class	445
Using Entity Builders to Create Test Data	448
Creating an Entity Builder	449
Entity Builder Examples	451
Creating New Builders	452

Part VIII

Guidewire ClaimCenter Configuration

33 Configuring Policy Behavior	461
Understanding Aggregate Limits	461
Aggregate Limit Data Model	462
Defining Aggregate Limits	463
Aggregate Limit Configuration	463
Storing Aggregate Limit Data	467
Aggregate Limit Used Recalculation	468
Advanced Aggregate Limit Configuration	468
Using the AggregateLimitTransactionPlugin	468
Example 1. Configuring Claims for Aggregate Limits	469
Example 2. Configuring Aggregate Limits for Deductible Recoveries	470
Specifying Policy Menu Links	471
Defining Internal ClaimCenter Policy Fields	471
34 Configuring Snapshot Views	473
How ClaimCenter Renders Claim Snapshots	473
Understanding Snapshot PCF Interaction	474
Encrypting Claim Snapshot Fields	474
Configuring Snapshot Templates	475
Snapshots and Data Model Extensions	475
35 Configuring Lines of Business	477
LOB Typelists	478
LOB Typelists and Policies	479
LOB Typelists and Incidents	479
Relationships Among LOB Typelists	479
Relationships Between LOB Typelists and Other Typelists	482
LOB Typelists Referenced by Non-LOB Typelists	483
LOB Typelists that Reference Non-LOB Typelists	483
Modal PCF Files that Use LOB Typelists	484

Editing LOB Typelists and Typecodes	484
LOB Typelists Editor Tabs	485
LOB Typelists Editor Right-click Menu.....	485
Editing an LOB Typelist	486
Editing an LOB Typecode	486
Adding a Child LOB Typecode	487
Adding a Non-LOB Typecode	487
Adding a Parent to an LOB Typecode	488
Removing an LOB Typecode	488
Retiring an LOB Typecode	488
Removing an LOB Typecode from Its Parent.....	488
Exporting an LOB Typecode or Typelist	489
Localizing an LOB Typelist	489
Coverages and Policies	489
Personal Auto Coverages and the LOB Typelists.....	489
Adding a New LossType Typecode	491
Adding a New ExposureType Typecode	492
36 Configuring Services	495
Importing Services	495
Vendor Service Tree	496
Vendor Service Details	498
Configuring Services – Steps	499
Editing Services	500
Localizing Services.....	500
Service Request Data Model.....	502
Lifecycle of a Service Request	503
Quote Only	504
Quote and Perform Service.....	505
Service Only	506
Unmanaged Service	506
Configuring Service Metrics.....	507
Example - Creating a Custom Service Request Metric.....	508
Configuring Service Request Metric Limits.....	509
37 Configuring Deductibles	511
Deductible Data Model.....	512
Typecodes for Deductibles	513
Permissions for Deductibles	513
Deductibles and Checks	513
Transferring Checks	513
Recoding Payments.....	514
Deleting and Voiding/Stopping Checks	514
Denying or Resubmitting Checks	514
Applying Deductibles on Multicurrency Checks	514
Cleared or Issued Checks	514
Cloning Checks.....	514
Deductibles and Rules	515
38 Configuring Weighted Workload Assignment	517
Enabling Weighted Workload	517
Weighted Workload Permissions	518
Workload Weight Recalculation	518
Closing Claims and Exposures	519
Weighted Workload Data Model	520

Weighted Workload Configuration	521
Workload Weight Computation	521
User Assignment API	522
Weighted Workload Assignment Strategies	523
Custom Configuration	524
Configuring the Default Weight in Code	524
Creating Custom Workload Strategies	524
Adding Criteria to Workload Classifications	525
Example - Add Flag Status to Claim Workload Classification	525
Adding Workload Classification Conditions	527
Example - Add a Color Condition to Claim Workload Classification	528
39 Working with Catastrophe Bulk Associations	535
Catastrophe Bulk Association Configuration	535
The GWCatastropheEnhancement Class	536
Catastrophes Data Model	536
Catastrophe Configuration Parameter	536
40 Configuring the Catastrophe Dashboard	537
Technical Design	537
Heat Map Generation and Components	538
Server, Browser, and Service Interactions	539
Principal Heat Map Classes and Files	540
Datasets and Map Data Points	541
Datasets and Caching	541
Datasets and Filtering	542
DBA Scripts to Improve Oracle and SQL Server Performance	542
Configuring the Heat Map	544
Common Configuration Use Cases	545
Advanced Configuration	545
41 Configuring Duplicate Claim and Check Searches	549
Understanding the Gosu Templates	549
Duplicate Claim Search	550
Duplicate Check Search	551
42 Configuring Claim Health Metrics	553
Adding a New Tier	553
Adding a High-Risk Indicator	555
Adding a New Claim Metric	558
43 Configuring Recently Viewed Claims	563
Adding a Loss Date to the Recently Viewed Claim List	563
44 Configuring Incidents	567
Implicit Incidents	567
Explicit Incidents	568
To Create a New Incident Type	568
Quick Claim Configuration	568
Incidents Data Model	569
Gosu and Incidents	569
Entities and Typelists Related to Incidents	571
45 Configuring Claim Archiving	573
Archiving and the Domain Graph	573
The Root Info Entity	574
Archiving in Guidewire ClaimCenter	575

Archiving and Encryption.....	575
Selecting Claims for Archive Eligibility	576
Retrieving Archived Objects from the Command Line	577
Monitoring Archiving Activity.....	577
Configuring Archiving	578
Archiving-related Configuration Parameters	578
Archive Rules	579
Archive Events	580
Archive Work Queue	580
Archiving Plugins	581
46 Configuring Special Instructions.....	583
Creating Service Tiers	583
Special Handling Data Model.....	584
Adding Service Tiers.....	584
Configuring Email Notifications	585
Configuring Claim Headline Comments	585
Configuring Activity Patterns for Special Handling.....	586
47 Configuring Roles and Relationships.....	587
Adding Contact Roles.....	587
Defining Contact Roles.....	588
Defining Role Constraints.....	588
How Configuring Roles Impacts Entity Data and Types	591
Generated Role Methods.....	592
Entity Property for Exclusive or Exclusive & Required Role	593
Entity Array Key for Required or ZeroToMany Role	593
Avoiding Errors with Contact Properties	594
Adding a New Contact Role: an Example	594
Relationships Between Contacts.....	596
Adding a Bidirectional Contact Relationship: an Example	597
48 Configuring Multicurrency.....	601
Multicurrency in Financial Calculations.....	601
Multicurrency Data Model	602
Foreign Exchange Adjustments	604
Foreign Exchange Transactions and Calculated Values.....	605
Foreign Exchange Adjustments on Claims and Payments	605

Part IX Guidewire ClaimCenter Financials

49 Configuring ClaimCenter Financials.....	611
Overview of the Financials Data Model	612
Financial-related Typelists	613
Financial Transaction Statuses	614
Financial Configuration Parameters	615
Batch Processes Related to Checks and Payments	616
50 ClaimCenter Financial Calculations	619
Financial Calculation APIs	619
Understanding ClaimCenter Financial Calculations	620
Using the Predefined Financial Calculations	621
Using a FinancialsCalculator Object	621
Obtaining Calculated Amounts	621
Different Ways to Retrieve an Amount	622

Predefined Financial Calculations	623
Payment Calculations	623
Reserve Calculations.....	624
Recovery Calculations	625
Floating Financials Calculations.....	625
Predefined Reinsurance Calculations	627
Reinsurance.....	627
Reinsurance Reserves	628
Reinsurance Recoveries	628
Retrieving Transaction IDs.....	628
Forming Composite Custom Expressions.....	629
Creating Custom Financial Gosu Classes.....	629
Financial Calculations with a Negative Value	630
Eroding and Non-eroding Payments.....	631
Using Floating Financial Calculations	631
Example 1. Using Reserve Lines in Multiple Currencies.....	631
51 Creating ClaimCenter Financial Transactions	635
Setting Reserves	635
Setting Reserve Values	635
Creating Reserve Transactions Directly	637
Creating Checks and Payments by Using CheckCreator	637
Creating Recovery Transactions.....	639
Claim Objects	639
Exposure Objects	639
Creating Recoveries Directly	640
Creating Recovery Reserve Transactions.....	640
Setting Recovery Reserve Values.....	640
Creating Recovery Reserve Transactions Directly	641
52 Configuring ClaimCenter Financial Screens	643
Configuring the Financial Summary Screen.....	643
The Financials Summary Page	644
Configuring the Filter Drop-Down	644
Defining the Model Used by a Panel Set	646
Controlling the Display of the Financial Model	647
Configuring Reserve Behavior	649
Understanding How Configuration Impacts Reserves	649
Reserve Permissions and Authority Limits.....	653
Setting the Number of Reserve Items to Show	653
Configuring Checks and Payments.....	654
Understanding Checks and Payments	654
Permissions and Authority Limits That Apply to Payments.....	654
Configuring the Check Wizard Recurrence Settings	655
Check Recurrence Data Model	655
Configuring the Check Wizard's Default Payment Type	656
Configuring Financial Holds	656
Modifying the Automatic Setting of Coverage in Question	656
Modifying the Conditions for Applying Financial Holds	657
Modifying claimcost Initial Reserves	657
Configuring Bulk Invoice Payments.....	658
Overview of Bulk Invoices	658
The Bulk Invoices Data Model	659
Configuring the Bulk Invoices Feature	660

About ClaimCenter Documentation

The following table lists the documents in ClaimCenter 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 ClaimCenter 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 ClaimCenter.
<i>Upgrade Guide</i>	Describes how to upgrade ClaimCenter from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing ClaimCenter application extensions and integrations.
<i>New and Changed Guide</i>	Describes new features and changes from prior ClaimCenter 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 ClaimCenter. The intended readers are everyone who installs the application for development or for production.
<i>System Administration Guide</i>	Describes how to manage a ClaimCenter 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 ClaimCenter for a global environment. Covers globalization topics such as global locales, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who work with locales and languages.
<i>Rules Guide</i>	Describes business rule methodology and the rule sets in ClaimCenter 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 ClaimCenter 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 ClaimCenter 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.

Conventions in This Document

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

Support

For assistance with this software release, contact Guidewire Customer Support:

- At the Guidewire Resource Portal – <http://guidewire.custhelp.com>
- By email – support@guidewire.com
- By phone – +1-650-356-4955

part I

ClaimCenter Configuration Basics

Overview of ClaimCenter Configuration

This topic provides some basic information, particularly about the Guidewire ClaimCenter 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 27
- “How You Configure ClaimCenter” on page 28
- “Types of Application Environments” on page 29
- “Deploying Configuration Files” on page 29
- “Regenerating the Data Dictionary and Security Dictionary” on page 30
- “Managing Configuration Changes” on page 32

What You Can Configure

Application configuration files determine virtually every aspect of the ClaimCenter 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 ClaimCenter Application

You can extend the functionality of the application:

- Build new views of claims and other associated objects.
- Create or edit validators on fields in the application.

Modify the ClaimCenter Interface

You can configure the text labels, colors, fonts, and images that comprise the visual appearance of the ClaimCenter 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 claim adjusters.

Integrate with External Systems

You can integrate ClaimCenter 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 ClaimCenter

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 111.

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 ClaimCenter 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 ClaimCenter. 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 ClaimCenter has the following characteristics:

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

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 ClaimCenter 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 ClaimCenter 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* ClaimCenter `bin` directory. For example:
`gwcc build-tomcat-war-dbcp`
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:
`ClaimCenter/webapps/cc`
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, `ClaimCenter/dist`). The `build` script places the new `cc.war` or `cc.ear` file in this directory.
5. Copy the newly created `cc.war` file to the *production* `webapps` folder (for Tomcat). If using WebSphere or WebLogic, use the administrative console to deploy the `cc.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 149.
- To understand the *Security Dictionary* and the information it includes, see “Security Dictionary” on page 466 in the *Application Guide*.

Generating the Data and Security Dictionaries in HTML Format

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

```
gwcc regen-dictionary
```

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

```
ClaimCenter/build/dictionary/data  
ClaimCenter/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 *ClaimCenter/bin* directory:

```
gwcc regen-dictionary -DoutputFormat=xml
```

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

```
ClaimCenter/build/dictionary/data/entityModel.xml  
ClaimCenter/build/dictionary/data/entityModel.xsd  
  
ClaimCenter/build/dictionary/security/securityDictionary.xml  
ClaimCenter/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 Data and Security 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:

```
gwcc 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 105 in the *Installation Guide*.

Aspects of Regenerating the Security Dictionary

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

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

ClaimCenter does not write this file information to the database.

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

ClaimCenter does not base the *Security Dictionary* on the actual roles that you have in your database. Instead, ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 34
- “Approval Parameters” on page 36
- “Archive Parameters” on page 36
- “Assignment Parameters” on page 38
- “Batch Process Parameters” on page 39
- “Business Calendar Parameters” on page 39
- “Cache Parameters” on page 41
- “Claim Catastrophe Parameters” on page 44
- “Claim Health Indicator and Metric Parameters” on page 44
- “Clustering Parameters” on page 45
- “Database Parameters” on page 48
- “Deduction Parameters” on page 51
- “Document Creation and Document Management Parameters” on page 52
- “Domain Graph Parameters” on page 53
- “Environment Parameters” on page 53
- “Financial Parameters” on page 58
- “Geocoding Feature Parameters” on page 61
- “Globalization Parameters” on page 62
- “Integration Parameters” on page 65
- “Miscellaneous Bulk Invoice Activity Pattern Parameters” on page 68
- “Miscellaneous Financial Activity Parameters” on page 68

- “Miscellaneous Parameters” on page 69
- “PDF Print Settings Parameters” on page 71
- “Scheduler and Workflow Parameters” on page 73
- “Search Parameters” on page 74
- “Security Parameters” on page 76
- “Segmentation Parameters” on page 79
- “Service Request Parameters” on page 79
- “Spell Check Parameters” on page 80
- “Statistics, Team, and Dashboard Parameters” on page 80
- “User Interface Parameters” on page 82
- “Work Queue Parameters” on page 85

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:

`ClaimCenter/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 ClaimCenter 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/cc` directory. Any modification of files in this directory can cause damage to the ClaimCenter 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. ClaimCenter configuration parameters are case-insensitive. If a parameter does not appear in the file, Guidewire ClaimCenter uses the default value, if the parameter has one.

Adding Custom Parameters to ClaimCenter

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 ClaimCenter, 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 105.

Accessing Configuration Parameters in Gosu

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

```
gw.api.system.PLConfigParameters  
gw.api.system.CCConfigParameters
```

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. ClaimCenter 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 ClaimCenter, 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, ClaimCenter throws an error.

Adding Custom MIME Types

Adding a new MIME type (MIME stands for Multipurpose Internet Mail Extensions), such that ClaimCenter 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 "|".
ClaimCenter uses this information to map from MIME type to file extension and file extension to MIME type. If mapping from type to extension, ClaimCenter uses the first extension in the list. If mapping from extension to type, ClaimCenter uses the first <mimetype> entry containing that extension.
- The image, in the tomcat/webapps/cc/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.

Approval Parameters

Guidewire provides the following configuration parameters in the config.xml file related to approval activities.

For information on editing config.xml and setting configuration parameters, see "Working with Configuration Parameters" on page 34.

BulkInvoiceApprovalPattern

Name of the activity pattern to use if creating bulk invoice approval activities.

Required: Yes

PaymentApprovalPattern

Name of an approval activity pattern to use if creating payment approval activities.

Required: Yes

RecoveryApprovalPattern

Name of the activity pattern to use if creating recovery approval activities.

Required: Yes

RecoveryReserveApprovalPattern

Name of the activity pattern to use if creating recovery reserve approval activities.

Required: Yes

ReserveApprovalPattern

Name of the approval activity pattern to use if creating reserve approval activities.

Required: Yes

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 ClaimCenter 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 34.

See also

- “More Information on Archiving” on page 133 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`, ClaimCenter creates a non-unique index on the `ArchivePartition` column for `Extractable` entities.

Furthermore, if the `Extractable` entity is `Keyable`, ClaimCenter creates a unique index on the `ID` and `ArchivePartition` columns.

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

AssignClaimToRetriever

Specifies to whom ClaimCenter assigns a retrieved claim:

- `True` assigns the claim to the user who retrieved the claim.
- `False` assigns a retrieved claim to the original group and user who owned it.

If it is not possible to reassign the claim to the original user, ClaimCenter assigns the retrieved claim to the supervisor of the group. If ClaimCenter cannot reassign the retrieved claim to the original group, ClaimCenter assigns the claim to `defaultowner`.

Default: False

DaysClosedBeforeArchive

ClaimCenter bases archive eligibility on the `entity.DateEligibleForArchive` field. For an archivable entity to be eligible for archiving, its `DateEligibleForArchive` property must be a non-null date and time that is not later than the current system date and time. In general, ClaimCenter calculates the `DateEligibleForArchive` value using the following formula:

`DateEligibleForArchive = DaysClosedBeforeArchive (in days) + current system date`

Default: 30

DaysRetrievedBeforeArchive

Used by the implementation of the `IArchiveSource` plugin in the base configuration to set the `DateEligibleForArchive` field on Claim as it retrieves a claim from the archive store. ClaimCenter calculates the `DateEligibleForArchive` value using the following formula:

```
DateEligibleForArchive = DaysRetrievedBeforeArchive (in days) + current system date
```

Default: 100

PolicySystemArchivingEnabled

It is possible to include archived policies in the ClaimCenter Policy Select search and the FNOL wizard Find Policy search. However, the policy system must retrieve any archived policies before ClaimCenter can use them. Configuration parameter `PolicySystemArchivingEnabled` controls whether you can include archived policies in searches.

This configuration parameter has the following valid settings:

PolicySystemArchivingEnabled	Meaning
true	The policy search result from both FNOL and policy select excludes archived policies.
false	If you also check Include Archived Policies in FNOL policy search or in the policy select screen, ClaimCenter includes archived policies in the search result. ClaimCenter also does the following: <ul style="list-style-type: none">• Sets the Address, City, State, and postal code fields to blank for archived policies• Displays a Status column

Default: False

RestorePattern

Code of the activity pattern that ClaimCenter uses to create retrieval activities. Upon retrieving a claim, ClaimCenter creates two activities:

- One activity for the retriever of the claim
- One activity for the assigned user of the claim, if different from the retriever

Default: restore

SnapshotEncryptionUpgradeChunkSize

Limits the number of claim snapshots that ClaimCenter upgrades during a change to the encryption plugin or during a change to encrypted fields. Set this parameter to zero to disable the limit.

Default: 50000

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 34.

AssignmentQueuesEnabled

Whether to display the ClaimCenter 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

WeightedAssignmentEnabled

Whether to enable assignment load balancing.

Default: false

WeightedAssignmentGlobalDefaultWeight

The global default weight of all assignable entities that do not match any classifications.

Default: 10

Minimum: 0

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 34.

BatchProcessHistoryPurgeDaysOld

Number of days to retain batch process history before ClaimCenter 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 34.

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 ClaimCenter to manage the official designation of holidays. Guidewire retains this configuration parameter to facilitate upgrade from older versions of ClaimCenter.

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 34.

See also

- “Application Server Caching” on page 67 in the *System Administration Guide*

ActivityPatternCacheMaxDuration

Upper bound on how long caches of activity pattern entities are allowed to exist without refresh, measured in seconds.

Default: 86400

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 67 in the *System Administration Guide* for more information.

Default: 600

GlobalCacheActiveTimeMinutes

Time period (in minutes) in which ClaimCenter 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 ClaimCenter 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 67 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 ClaimCenter collects to explain why items are evicted from the cache. ClaimCenter 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 ClaimCenter Management Beans page to enable the collection of detailed statistics for the global cache. If you disable the `GlobalCacheDetailedStats` parameter, ClaimCenter does not display the `Evict Information` and `Type of Cache Misses` graphs.

Default: False

GlobalCacheReapingTimeMinutes

Time (in minutes) since the last use before ClaimCenter considers cached items to be eligible for reaping. You can think of this as the period during which ClaimCenter is most likely to reuse an entity. See “Application Server Caching” on page 67 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 67 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 67 in the *System Administration Guide* for more information.

Default: 15

Set for Environment: Yes

GlobalCacheStaleTimeMinutes

Time (in minutes) since the last write before ClaimCenter considers cached items to be stale and thus eligible for reaping. See “Application Server Caching” on page 67 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 ClaimCenter preserves statistics for historical comparison. See “Application Server Caching” on page 67 in the *System Administration Guide* for more information.

Default: 7

Minimum: 2

Maximum: 365

Set for Environment: Yes

GlobalCacheStatsWindowMinutes

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

- To define the period of time to preserve the reason for which ClaimCenter evicts an item from the cache, after the event occurred. If a cache miss occurs, ClaimCenter 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 67 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 67 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 67 in the *System Administration Guide* for more information.

Default: 600

TreeViewRefresh

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

Default: 120

ZoneCacheRefreshIntervalSecs

The time between refreshes of the zone cache, in seconds. See “Application Server Caching” on page 67 in the *System Administration Guide* for more information.

Default: 86400

Claim Catastrophe Parameters

Before you can use the **Catastrophe Search** page and its catastrophe heat map, you must enable the functionality. For details, see “Enabling Catastrophe Search and Heat Maps” on page 18 in the *System Administration Guide*.

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

HeatMapCredential

Set this configuration parameter to the value you obtained during the licensing process for the Bing Maps Ajax Control.

Default: `None`

HeatMapServiceTemplate

Set to the fully qualified path to the class that manages the Bing map configuration.

Default: `gw.api.heatmap.BingMap`

MaxCatastropheClaimFinderSearchResults

Maximum number of claims that ClaimCenter relates to a single catastrophe in the `CatClaimFinder` batch process, (used to find catastrophe-related claims). See “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide* for a discussion of ClaimCenter batch processes.

Default: 1000

Claim Health Indicator and Metric Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

ClaimHealthCalcMaxLossDateInYears

Maximum number of years to look back to find claims on which to calculate metrics and indicators. This parameter strictly limits the number of claims found for the `ClaimHealthCalculation` batch process.

Default: 2

InitialReserveAllowedPeriod

Number of days that new initial reserves contribute to the initial reserve sum of the Percent * Reserve Change Claim Metric calculation after ClaimCenter creates the first initial reserve. An initial reserve is a reserve that ClaimCenter creates during creation of a claim or exposure. It is also the first set of reserves that ClaimCenter creates on the claim or exposure if there are no previous reserves for those entities.

Default: 3

MaxClaimResultsPerClaimHealthCalcBatch

Maximum number of claims for each invocation of the `ClaimHealthCalculation` batch process to calculate metrics and indicators. This parameter strictly limits the number of claims that can be processed at a single time. The `ClaimHealthCalculation` batch process calculates the claim metrics and indicators for found claims.

Default: 1000

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 ClaimCenter servers in a configuration known as a cluster. A cluster distributes client connections among multiple ClaimCenter 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 34.

See also

- “Clustering Application Servers” on page 79 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):" +
    "MERGE2(max_interval=30000;min_interval=10000):" +
    "FD(timeout=5000;max_tries=5):" +
    "VERIFY_SUSPECT(timeout=3000;num_msgs=3):" +
    "pbcast.NAKACK(retransmit_timeout=600,1200,2400,4800;discard_delivered_msgs=true):" +
    "UNICAST(timeout=600,1200,2400,4800):" +
    "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

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 34.

DisableHashJoinForClaimSearch

Guidewire provides a workaround for query plan problems related to hash joins that occur if executing certain claim searches on Oracle. This parameter controls part of the workaround. The parameter has no effect on databases other than Oracle. See also `DisableSortMergeJoinForTeamGroupActivities`.

Default: True

DisableHashJoinForTeamGroupActivities

ClaimCenter works around hash join-related query plan problems while executing the `team group activities` page's main query on Oracle. This parameter controls part of the work around and is `true` by default. This parameter has no effect on databases other than Oracle.

Default: True

DisableIndexFastFullScanForClaimSearch

ClaimCenter works around some bad execution plans by disabling index fast full scan while executing certain claim searches on Oracle. This parameter controls the work around and is `true` by default. If a future version of Oracle fixes the defect, you can safely remove this parameter. The parameter affects the Oracle database only.

Default: True

DisableIndexFastFullScanForRecoverySearch

ClaimCenter works around some bad execution plans by disabling index fast full scan while executing certain recovery searches on Oracle. This parameter controls the work around and is `true` by default. If a future version of Oracle fixes the defect, you can safely remove this parameter. The parameter affects the Oracle database only.

Default: True

DisableIndexFastFullScanForTeamGroupActivities

ClaimCenter works around index fast full scan related query plan problems while executing the `Team Group Activities` page's main query on Oracle. This parameter controls the work around and is `true` by default. If a future

version of Oracle fixes the defect, you can safely remove this parameter. The parameter has no effect on databases other than Oracle.

Default: True

DisableSortMergeJoinForClaimSearch

Guidewire provides a work-around for sort-merge join query plan problems that occur if executing certain claim searches on Oracle. This parameter controls part of the work-around if `DisableHashJoinForClaimSearch` is also set to `true`. The parameter has no effect on databases other than Oracle.

Default: True

DisableSortMergeJoinForTeamGroupActivities

ClaimCenter works around sort merge join query plan problems while executing the `team group activities` page's main query on Oracle. This parameter controls part of the workaround if the value of `DisableHashJoinForClaimSearch` is set to `true`. It is `true` by default. The 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

IdentifyQueriesViaComments

(SQL Server and DB2) Whether to provide comments with contextual information in certain SQL `Select` statements sent to the relational database.

The SQL comments are in the format:

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

The `applicationName` component of the comment is `ClaimCenter`.

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

Default: False

See also

- “Enabling Context Comments in Queries on SQL Server or DB2” on page 173 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

QueryRewriteForClaimSearch

It is possible to create materialized views in an Oracle schema to improve the performance of queries that ClaimCenter runs as part of a Claim search operation. Materialized views can be useful if performing a search for a claimant or for any involved party using the name of a person or a company. If you implement materialized

views in the ClaimCenter schema, then Oracle attempts to use these materialized views if a re-written query block matches the text defined in the view.

Guidewire provides configuration parameter `QueryRewriteForClaimSearch` to enable various options for an Oracle query re-write using materialized view. By setting this parameter, you can force a query to be rewritten using a materialized view or to let the Oracle optimizer make the choice based on the cost calculation.

The following list describes the valid values for this parameter:

Value	Meaning
FORCE/STALE	Oracle attempts to rewrite the query using an appropriate materialized view even if the optimizer cost estimate is high. Oracle allows the rewrite even if the data in the materialized is not the same as in the base tables.
FORCE/NOSTALE	Oracle attempts to rewrite the query using an appropriate materialized view even if the optimizer cost estimate is high. Oracle ignores the materialized view if the data in the view is not fresh.
COST/STALE	If the Oracle cost-based optimizer evaluates the rewrite to be cheaper than other plans, it uses the materialized view. If it is costlier to execute the rewritten path, then Oracle performs a join of the base tables. The rewrite can happen even if the data in the view is stale.
COST/NOSTALE	If the Oracle cost-based optimizer evaluates the rewrite to be cheaper than other plans, it uses the materialized view. If it is costlier to execute the rewritten path, then Oracle performs a join of the base tables. If the data in the view is not fresh, Oracle ignores the view and performs the join on the base tables.

Note: If you provide an invalid value, the server ignores it.

Default: None

Disabling Query Rewrites

If you implement materialized views in the ClaimCenter schema, then Oracle attempts to use these materialized views if a re-written query block matches the text defined in the view. However, the use of materialized views in database queries is not always desirable due to performance considerations.

Thus, ClaimCenter provides an option to disable the rewriting of queries using materialized views. You can disable the use of materialized views in Oracle database queries by setting parameter `queryRewriteEnabled` to `false` in the `<database>` element in `config.xml`. For example:

```
<param name="queryRewriteEnabled" value="false"/>
```

The only valid value for the parameter is `false`:

- If you set this parameter to `false` in `config.xml`, ClaimCenter runs the following statement for each query session:


```
ALTER SESSION SET QUERY_REWRITE_ENABLED = FALSE;
```
- If you do not set this parameter in `config.xml` (meaning that you remove the parameter entirely from `config.xml`), then Oracle attempts to use any available materialized view in database queries.

Materialized Views

For a description of how to create materialized views in a ClaimCenter schema, consult the following Guidewire white paper, section 19 (Materialized Views):

ClaimCenter 6.0 using Oracle Database

You can find this document at the following location on the Guidewire Resource Portal:

<https://guidewire.custhelp.com/app/resources/infrastructure/documents>

The (Server Tools) Info Pages → Oracle AWR Information page is aware of materialized views. You can use the information on this page to troubleshoot performance problems with the view. However, if the view is refreshed on demand, then the Oracle AWR Information page does not capture the refresh queries.

Stale Data

In performance testing, Guidewire observed significant performance degradation if the materialized view was configured to refresh on commit. This is due to a synchronization enqueue required by the refresh process. However, any refresh of the data done outside of the commit operation can potentially display stale data during the search.

Oracle uses a cost-based optimizer approach to determine whether to use a materialized view for a given query. It also expects the data to be fresh for the rewrite. As the refresh process is based on the number of changes to contact and claim contacts, Guidewire strongly recommends that you schedule the refresh process accordingly.

See also

- “Configuring a Database Connection” on page 62 in the *Installation Guide*
- “Setting Search Parameters for Oracle” on page 39 in the *System Administration Guide*

SetSemiJoinNestedLoopsForClaimSearch

(Oracle) ClaimCenter works around semi-join query plan problems by forcing nested loop semi-join instead of choose while executing certain claim searches on Oracle. This parameter controls part of the work around and is true by default. This parameter has no effect on databases other than Oracle.

Default: True

Deduction Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

BackupWithholdingTypeCode

The typecode in the `DeductionType` typelist for backup withholding. The default is `irs`. You must define this parameter for the backup withholding plugin to work. Also, this parameter must also correspond to a valid `DeductionType` typecode.

Default: `irs`

CalculateBackupWithholdingDeduction

Whether ClaimCenter calculates backup withholding for applicable checks.

Default: True

StandardWithholdingRate

Standard backup withholding rate, as defined by the U.S. Internal Revenue Service, for use by the backup withholding plugin. The number is a percentage. (For example, 28.0 means 28.0 percent.) The backup withholding plugin does not work if you do not define this parameter.

Default: 28.0

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 151 in the *System Administration Guide*.
- For information on editing config.xml and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

AllowDocumentAssistant

Whether to allow document management controls in the ClaimCenter 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 ClaimCenter 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 ClaimCenter uses to validate document template descriptor XML files. Specify this location relative to the following directory:

`modules/configuration/config/resources/doctemplates`

MaximumFileUploadSize

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 ClaimCenter **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 `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

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 ClaimCenter 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 `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

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 54.

Default: false

AlwaysShowPhoneWidgetRegionCode

Determines whether the phone widget operates in multi- or single-region mode.

Default: true

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, ClaimCenter 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 ClaimCenter 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 249 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 254 in the *Integration Guide*.
- For information on using the Plugins Registry editor, see “Using the Plugins Registry Editor” on page 113.

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 53

EnableInternalDebugTools

Make internal debug tools available to developer.

Default: False

Set for Environment: Yes

EntityValidationOrder

Order in which to execute validation if validating multiple entities. ClaimCenter validates all other validatable entities not specified in this list after all listed entities, in no particular order.

Note: Guidewire does not guarantee that ClaimCenter validates entities of a given type (such as the exposures on a claim) in any deterministic order with respect to one another.

Default: Policy, Claim, Exposure, Matter, TransactionSet

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 ClaimCenter 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 56.

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`, ClaimCenter 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 56.

StrictDataTypes

Controls whether ClaimCenter uses the pre-ClaimCenter 6.0 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 useful for new ClaimCenter installations that want to implement the new behavior.

StrictDataTypes = true

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

- 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 ClaimCenter user interface and in the field setter.
- Validates numeric range constraints in both the ClaimCenter user interface and in the field setter.

StrictDataTypes = false

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

- 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, ClaimCenter 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 ClaimCenter.

Default: su

UseOldStylePreUpdate

If set to `true`, then changes to an entity trigger execution of the existing Preupdate and Validation rules during a bundle commit of the entity. (That is, as long as the entity implements the `Validatable` delegate.) If set to `false`, then ClaimCenter invokes the `IPreUpdateHandler` plugin on the bundle commit.

Default: True

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 ClaimCenter works with monetary amounts.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

See also

- “Claim Financials” on page 287 in the *Application Guide*
- “Configuring ClaimCenter Financials” on page 611
- “ClaimCenter Financial Calculations” on page 619
- “Configuring Multicurrency” on page 601
- “Configuring Currencies” on page 85 in the *Globalization Guide*

AllowMultipleLineItems

Whether to allow multiple line items in a transaction. See also `UseDeductibleHandling`.

Default: True

AllowMultiplePayments

Whether to allow a single check to reflect multiple payments.

Default: True

AllowNoPriorPaymentSupplement

Whether to allow a user to create supplemental payments on a closed claim or exposure with no prior payments.

Default: False

AllowPaymentsExceedReservesLimits

If `true`, a user can submit payments that exceed available reserves up to the amount limited by the `paymentsexceedreserves` authority limits. Otherwise, ClaimCenter does not allow partial or final payments that exceed reserves.

Default: False

CheckAuthorityLimits

Controls whether ClaimCenter checks authority during approval processing of a `TransactionSet`.

Default: True

CloseClaimAfterFinalPayment

If `true`, ClaimCenter attempts to automatically close a claim if a final payment closes the last open exposure.

Default: True

CloseExposureAfterFinalPayment

If `true`, ClaimCenter attempts to automatically close the exposure of a Final payment when that payment's Check is escalated.

Default: True

EnableMulticurrencyReserving

Whether to enable multicurrency reserving support. Note that this parameter is not intended to be used by non-Gosu classes, or to be accessed directly; use `CCConfigUtil.EnableMulticurrencyReserving` instead.

Default: false

Set for Environment: Yes

ExchangeRatesCacheRefreshIntervalSecs

This parameter sets the time between refreshes of the `ExchangeRateSet` cache, in seconds.

Default: 600

Financials

Specifies the level of financials functionality that is available in the application. Available options are `view` for read-only values or `entry` to enable editing the financial values directly in ClaimCenter.

ClaimCenter supports the following values:

entry	financials entry
view	financials view (read-only)

Default: entry

PaymentLogThreshold

ClaimCenter logs payments greater than this threshold. This integer value must be zero (0) or greater.

Default: 500

Can Change on Running Server: Yes

PaymentRoundingMode

Use to set the rounding mode for conversion of amounts among `TransactionAmount`, `ClaimAmount`, and `ReportingAmount` columns on the `TransactionLineItem` entity for Payment transactions. 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:

- DOWN
- HALF_UP
- HALF_EVEN
- HALF_DOWN

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

```
gw.api.util.CCCurrencyUtil.getRoundingMode(Payment)
```

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

Default: DOWN

Permanent: Yes

See also

- “Choosing a Rounding Mode” on page 92 in the *Globalization Guide*

ReserveRoundingMode

Use to set the rounding mode for conversion of amounts among `TransactionAmount`, `ClaimAmount`, and `ReportingAmount` columns on the `TransactionLineItem` entity for Reserve transactions.

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:

- UP
- HALF_UP
- HALF_EVEN

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

```
gw.api.util.CCCurrencyUtil.getRoundingMode(Reserve)
```

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

Default: UP

Permanent: Yes

See also

- “Choosing a Rounding Mode” on page 92 in the *Globalization Guide*

SetReservesByTotalIncurred

Specifies the way in which you modify reserves in the ClaimCenter interface.

- If set to `true`, the user can set the **Total Incurred** values
- If set to `false`, the user can set the **Available Reserves** values.

Default: False

UseDeductibleHandling

Whether to use Deductible Handling. If this value is `true`, then `AllowMultipleLineItems` must be `true` as well.

Default: True

See also

- “Deductible Handling” on page 349 in the *Application Guide*

UseRecoveryReserves

Whether to use recovery reserve transactions in financial calculations.

Default: True

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 34.

UseGeocodingInPrimaryApp

If `true`, ClaimCenter enables proximity search for users in the assignment user interface. ContactManager does not respond to this parameter.

Default: False

UseGeocodingInAddressBook

Set to `true` if you have ClaimCenter integrated with ContactManager and ContactManager has geocoding enabled for vendors. This setting enables vendor search in the ClaimCenter and ContactManager user interfaces.

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

IMPORTANT If the setting for this configuration parameter differs between Contact Manager and ClaimCenter, it is possible for the application to display distance-related messages incorrectly.

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. This parameter does not have to match the value of the corresponding parameter in the ContactManager `config.xml` file.

Default: 1000

UseMetricDistancesByDefault

If `true`, ClaimCenter uses kilometers and metric distances instead of miles and United States distances for driving distance or directions. Set this parameter identically in both ClaimCenter 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 34.

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 28 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 71 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 ClaimCenter 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 *reporting 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 91 in the *Globalization Guide*

DefaultRoundingMode

Sets the default rounding mode for money and currency 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 92 in the *Globalization Guide*
- “PaymentRoundingMode” on page 60
- “ReserveRoundingMode” on page 60

MulticurrencyDisplayMode

Determines whether ClaimCenter displays currency selectors for monetary values. You can set MulticurrencyDisplayMode to one of the following values:

- SINGLE
- MULTIPLE

In the base configuration of ClaimCenter, 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 ClaimCenter 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 92 in the *Globalization Guide*.

DefaultCountryCode

The default ISO country code to use if the country for address is not set explicitly. ClaimCenter 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 34.

ContactAutoSyncWorkItemChunkSize

If you integrate Guidewire ClaimCenter with Guidewire ContactManager, it is necessary to maintain the synchronization of contacts between the two Guidewire applications. Batch process `ContactAutoSync` controls this synchronization. (See “ContactAutoSync” on page 133 in the *System Administration Guide* for details.)

It is common to have a large number of local instances of each contact in Guidewire ClaimCenter, one for each claim that uses that contact. During contact synchronization between ClaimCenter and ContactManager, ClaimCenter processes the table for highly linked contacts by dividing the contents of the contact table into smaller groups of contacts. (This process is known as chunking as the end result is chunks of data.) ClaimCenter then creates a work item to process each chunk of contacts. Parameter `ContactAutoSyncWorkItemChunkSize` specifies the maximum number of contacts that each single `ContactAutoSyncWorkItem` is to process, or, in other words, the size of the chunk.

Note: Parameter `ContactAutoSyncWorkItemChunkSize` is meaningful only if Guidewire ClaimCenter is integrated with Guidewire ContactManager.

Default: 400

DefaultXmlExportEncryptionId

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

Default: null (no encryption)

EnableMetroIntegration

Whether to enable Metropolitan Reporting Bureau integration. If `true`, there is a working integration that sends messages from ClaimCenter to the Metropolitan Reporting Bureau service (requesting Metropolitan reports).

Default: False

Set for Environment: Yes

InstantaneousContactAutoSync

Whether to process contact automatic synchronization in ClaimCenter at the time of receiving the notification:

- If you set InstantaneousContactAutoSync to `false`, ClaimCenter synchronizes contacts only while running the `ContactAutoSync` batch process.
- If you set InstantaneousContactAutoSync to `true`, the `ContactAutoSync` worker activates automatically as ClaimCenter receives autosync events and immediately updates all ClaimCenter copies of a contact stored in `ContactManager`.

See also

- For a description of this parameter as it relates to contact synchronization, see “ClaimCenter Synchronizing Controls” on page 201 in the *Contact Management Guide*.
- The full description of this parameter is under “Running and Scheduling the Work Queue” on page 202 in the *Contact Management Guide*.

Default: `True`

ISOPropertiesFileName

Name of the ISO properties file in the `ClaimCenter/modules/cc/config/iso` configuration directory.

Default: `ISO.properties`

Set for Environment: Yes

KeepCompletedMessagesForDays

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

Default: `90`

LoadSoapServicesOnStartup

If true, for RPCE web services, instantiation of all the implementation classes and registering of the WSDL happens on server startup. Otherwise, initialization of all implementation classes happens on the first request to any service. See “Configuring When To Initialize RPCE Web Service Implementation Classes” on page 99 in the *Integration Guide*.

Default: `false`

LockPrimaryEntityDuringMessageHandling

If it is set to `true`, ClaimCenter 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`

MetroPropertiesFileName

Name of the Metropolitan properties file in the `ClaimCenter/modules/cc/config/metro` configuration directory. ClaimCenter uses this files to set up fields in the XML messages sent to the Metropolitan Reporting Bureau. See `EnableMetroIntegration` as well.

Default: `Metro.properties`

Set for Environment: Yes

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

PolicySystemURL

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

- If integrating Guidewire ClaimCenter with Guidewire PolicyCenter, then set this parameter to the PolicyCenter base URL (for example, `http://server/pc`). In this case, the exit points add the appropriate PolicyCenter entry point.
- If integrating with a non-Guidewire policy 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 ClaimCenter hides the buttons in the interface that take you to the exit points.

Default: Empty string

UseSafeBundleForWebServicesOperations

Note: The described bundle behavior affects RPC-Encoded web services only. With WS-I web services, bundle management is under the service author's control. For WS-I web services, use the `gw.transaction.Transaction.runWithNewBundle` block syntax instead. See “Running Code in an Entirely New Bundle” on page 342 in the *Gosu Reference Guide* for details.

Configuration parameter `UseSafeBundleForWebServiceOperations` changes the behavior of bundle commits in RPC-Encoded web services published on this server. The default value is `true`.

- If set to `false`, the application ignores bean version conflicts as it commits a bundle.
- If set to `true`, the application detects (and does not ignore) bean version conflicts.

If you set this parameter to `true`, it is possible for Gosu to throw a `ConcurrentDataChangeException` exception. (The exception text actually reads “*Database bean version conflict*” or similar.) This can happen if another thread or cluster node modified this entity as it was loaded from the database. If this error condition occurs, then

the SOAP client receives a `SOAPRetryableException`. Guidewire strongly recommends that web service clients catch all retryable exceptions such as this and retry the SOAP API call.

IMPORTANT A value of `true` for `UseSafeBundleForWebServiceOperations` sets the behavior system-wide.

IMPORTANT A previous workaround for this issue used the `setIgnoreVersionConflicts` method on the bundle at the beginning of SOAP implementation methods. If you used this workaround, then you must update your client-side logic for detecting and retrying the SOAP call in the case of a concurrent change exception.

Default: True

Miscellaneous Bulk Invoice Activity Pattern Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

`BulkInvoiceItemValidationFailedPattern`

Name of the activity pattern to use in creating an activity to alert about a failure during processing of a bulk invoice item.

Required: Yes

`BulkInvoiceUnableToStopPattern`

Name of the activity pattern to use if creating an activity to alert that ClaimCenter was unable to stop a bulk invoice. More technically, it was not possible to update the status from `Pending-stop` or `Stopped` to `Issued` or `Cleared`.

Required: Yes

`BulkInvoiceUnableToVoidPattern`

Name of the activity pattern to use in creating an activity to alert that ClaimCenter was unable to void a bulk invoice. More technically, it was not possible to update the status from `Pending-void` or `Voided` to `Issued` or `Cleared`.

Required: Yes

Miscellaneous Financial Activity Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to financial activity patterns.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

CheckDeniedPattern

Name of the activity pattern to use if creating an alert that a down-stream system has denied a check.

Required: Yes

CheckUnableToStopPattern

Name of the activity pattern to use if creating an alert that ClaimCenter cannot stop a check.

Required: Yes

CheckUnableToVoidPattern

Name of the activity pattern to use if creating an alert that ClaimCenter cannot void a check.

Required: Yes

LastPaymentReminderPattern

Name of the activity pattern to use if creating an alert to signal the approach of the last payment in a set of recurrence checks.

Required: Yes

RecoveryDeniedPattern

Name of the activity pattern to use if creating an alert that a down-stream system has denied a recovery.

Required: Yes

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 34.

ClaimLossDateDemarcation

Default time for a claim’s loss date when it is not specified in the FNOL. Set this to match the default time at which the policy administration system sets the effective date of the policy.

Default: 12:00 AM

Set for Environment: Yes

ConsistencyCheckerThreads

Number of threads to use when running the consistency checker.

Default: 1

EnableClaimNumberGeneration

Whether to enable automatic claim number generation (through an external plugin). If you enable claim number generation, then you must also provide an external Claim Number Generator plugin. If enabled, claim number

generation must succeed in order for a claim to be added through either the New Claim wizard or the integration tools. This does not affect claims added through staging tables.

Default: True

EnableClaimSnapshot

Whether to create snapshots for imported and created claims. The claim snapshot contains a version of the claim data before any automated processing by ClaimCenter.

Default: True

EnableStatCoding

Whether to enable statistical coding support.

Default: True

JGroupsClusterChannel

Whether to use JGroups based cluster channel.

Default: true

ListViewPageSizeDefault

The default number of entries that ClaimCenter 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

MaintainPolicyValidationLevelOnPolicyChange

If **true**, any time that you change or refresh the policy for a claim, ClaimCenter validates the new policy at the level of the old policy. If **false**, ClaimCenter validates the new policy at the **newloss** level. In either case, a validation failure causes ClaimCenter to revert the policy refresh or change.

Default: True

MaxCachedClaimSnapshots

Limits the number of claim snapshots that ClaimCenter caches in memory. This integer value must be zero (0) or greater, but less than ten (10).

Default: 3

Minimum: 0

Maximum: 10

MaxStatCodesInDropdown

Maximum number of statistics codes to show in the statistics code picker drop-down.

Default: 20

ProfilerDataPurgeDaysOld

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

Default: 30

VendorNotificationAPIRetryTime

Amount of time (in seconds) to wait before attempting to retry a Vendor Notification Message that failed validation.

Default: 10

Set For Environment: Yes

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 ClaimCenter.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

DefaultContentDispositionMode

The `Content-Disposition` header setting to use any time that ClaimCenter returns document content directly to the browser. ClaimCenter 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, ClaimCenter 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, ClaimCenter generates an error.

Default: -1

PrintPageHeight

Total print height of page.

Default: 8.5in

PrintPageWidth

Total print width of page.

Default: 11in

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.

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 34.

IdleClaimThresholdDays

ClaimCenter schedules claims that have not been touched (including edits or exception checks) for this many days for exception detection. This integer value must be zero (0) or greater.

Default: 7

Can Change on Running Server: Yes

SchedulerEnabled

Whether to enable the internal batch process application scheduler. See “Batch Processes and Work Queues” on page 123 in the *System Administration Guide* for more information on batch processes and the scheduler.

Default: True

Can Change on Running Server: Yes

SeparateIdleClaimExceptionMonitor

If `true`, run exception monitoring rules for idle cases at a separate time.

Default: `True`

WorkflowLogDebug

Configuration parameter `WorkflowLogDebug` takes a Boolean value:

- If set to `true`, ClaimCenter outputs the ordinary verbose system workflow log messages from the Guidewire server to the workflow log.
- If set to `false`, ClaimCenter 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 ClaimCenter deletes it.

Default: 30

WorkflowPurgeDaysOld

Number of days to retain workflow information before ClaimCenter 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 34.

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 → Claims → Search by Contact` screen, which uses free-text search.

Default: `False`

See also

- “Free-text Search Configuration” on page 360

MaxActivitySearchResults

Maximum number of activities that ClaimCenter returns in a search. If the number to return is greater than this value, ClaimCenter prompts the user to narrow the search parameters. This integer value must be one (1) or greater.

Default: 300

MaxBulkInvoiceSearchResults

Maximum number of bulk invoices that ClaimCenter returns in a search. If the number to return is greater than this value, ClaimCenter prompts the user to narrow the search parameters. This integer value must be one (1) or greater.

Default: 300

MaxCheckSearchResults

Maximum number of checks that ClaimCenter returns in a search. If the number to return is greater than this value, ClaimCenter prompts the user to narrow the search parameters. This integer value must be one (1) or greater.

Default: 300

MaxClaimSearchResults

Maximum number of results that ClaimCenter returns for a claim search. This integer value must be one (1) or greater. If the number of results to return is greater than this value, ClaimCenter prompts the user to narrow the search parameters.

Default: 300

MaxContactSearchResults

Maximum number of contacts that ClaimCenter returns in a search. If the number to return is greater than this value, then ClaimCenter prompts the user to narrow the search parameters. This integer value must be one (1) or greater.

Default: 100

Default: 300

MaxDocTemplateSearchResults

Maximum number of document templates that ClaimCenter returns in a search. If the number to return is greater than this value, then ClaimCenter prompts the user to narrow the search parameters. This integer value must be one (1) or greater.

Default: 50

MaxDuplicateContactSearchResults

Maximum number of duplicate results to return from a contact search. This integer value must be zero (0) or greater.

Default: 25

MaxNoteSearchResults

Maximum number of notes that ClaimCenter returns in a search. If the number to return is greater than this value, ClaimCenter prompts the user to narrow the search parameters. This integer value must be zero (0) or greater. A value of zero indicates that there is no limit on the search.

Default: 25

MaxPolicySearchResults

Maximum number of policies that ClaimCenter returns in a search. If the number to return is greater than this value, then ClaimCenter prompts the user to narrow the search parameters. This integer value must be one (1) or greater.

Default: 25

MaxRecoverySearchResults

Maximum number of policies that ClaimCenter returns in a search. If the number to return is greater than this value, then ClaimCenter prompts the user to narrow the search parameters. This integer value must be one (1) or greater.

Default: 300

SetSemiJoinNestedLoopsForClaimSearch

Guidewire provides a work-around for semi-join query plan problems by forcing nested loop semi-join queries while executing certain claim searches on Oracle. This parameter controls part of the work-around. The parameter has no effect on databases other than Oracle.

Default: True

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 34.

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 ClaimCenter 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 ClaimCenter 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

MaxACLParameters

Maximum number of users and groups to directly include in search queries that check the claim access control list. Beyond this maximum limit, ClaimCenter stores users and groups in database tables. You must then use an additional join in the query to check the claim access control list.

Checking the claim access control list can involve a large number of groups and users. For example, if `EnableDownlinePermissions` is `true`, someone who supervises many groups and users has access to control lists that contain any of their supervisees. Including all these groups and users in the query can be done directly by including them as parameters to the query. Or, you can store them in database tables and doing extra joins in the query.

For small numbers of groups and users, direct parameters are the best choice. For large numbers, in the range of thousands, the extra join can be better. This parameter, `MaxACLParameters`, determines the point at which the query code switches from using direct parameters to using extra joins.

The `MaxACLParameters` can take the following values:

- **A value of -1 or less** – Instructs ClaimCenter to use the appropriate default for the current database. Thus, ClaimCenter chooses the best value, as determined by Guidewire performance testing, for the current type of database.
- **A value of 0** – Instructs ClaimCenter to always use parameters, and to never use a join in a query. This works even for very large numbers of groups and users, 3000 or more, on an Oracle database. However, it is not suitable for the SQL Server database, which limits the total number of parameters to 2100.
- **A value of +1 or greater** – Instructs ClaimCenter to use that value as a threshold. If the number of groups and users is less than the threshold, then a query uses parameters. If the number is larger the threshold, a query uses database tables and extra joins. Guidewire strongly recommends that you do not use a positive value for the Oracle database. This is because the Oracle database can cope with large numbers of parameters, but tends to choose very bad query plans for the extra joins.

In summary, Guidewire recommends that most ClaimCenter installations use the default value of -1, which chooses the best value for the current database type.

Database	Notes
SQL Server	For those ClaimCenter installations that use SQL Server as the database, Guidewire recommends the following: <ul style="list-style-type: none">• Do not set this value to 0.• Do not set it to any value greater than approximately 2000 due to the risk of hitting the 2100 parameter limit.
Oracle	For those ClaimCenter installations that use the Oracle database. Guidewire expressly recommends that you do not use positive values due to the risk of bad query plans.

Default: -1

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 ClaimCenter does not require a password. (Guidewire does not recommend this.)

Default: 8

Minimum: 0

RestrictContactPotentialMatchToPermittedItems

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

Default: True

RestrictSearchesToPermittedItems

Whether ClaimCenter 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

ShouldSyncUserRolesInLDAP

If **True**, then ClaimCenter synchronizes contacts with the roles they belong to after authenticating with the external authentication source.

Default: False

UseACLPermissions

Whether to use the ACL permission model.

- If **false**, the privilege that a user holds applies to every claim.
- If **true**, the `ClaimAccess` table controls claim access.

Default: True

Segmentation Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to claim and exposure segmentation.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

ClaimSegment

Default value to set the `Segment` field to on a claim, if ClaimCenter cannot determine another segment.

Required: Yes

ClaimStrategy

The default value to set the `Strategy` field to on a claim, if ClaimCenter cannot determine another strategy.

Required: Yes

ExposureSegment

Default value to set the `Segment` field to on an exposure, if ClaimCenter cannot determine another segment.

Required: Yes

ExposureStrategy

Default value to set the `Strategy` field on an exposure, if ClaimCenter cannot determine another strategy.

Required: Yes

Service Request Parameters

Guidewire provides the following configuration parameters in the `config.xml` file that relate to claim and exposure segmentation.

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

ServiceRequestAPIMaxDaysKeepActiveWithoutInvoice

Maximum number of days that a service request will be considered active by the `ServiceRequestAPI` when it is work complete but has no valid invoices.

Default: 90

Set For Environment: Yes

ServiceRequestAPIMaxMessageResults

Maximum number of service request messages returned through the `ServiceRequestAPI` web service.

Default: 50

Set For Environment: Yes

ServiceRequestAPIMaxSearchResults

Maximum number of service requests returned by a search performed through the ServiceRequestAPI web service.

Default: 250

Set For Environment: Yes

Spell Check Parameters

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

For information on editing config.xml and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

CheckSpellingOnChange

If true, navigating away from a field enabled for spell checking—after making any change—invokes the spelling checker for the field.

Default: False

CheckSpellingOnDemand

If true, a **Check Spelling** button appears in the toolbar of any screen that contains editable fields, if the screen is enabled for spell checking.

Default: False

Statistics, Team, and Dashboard Parameters

Guidewire provides the following configuration parameters in the config.xml file that relate to ClaimCenter statistics, the Team tab, and the Dashboard.

For information on editing config.xml and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

AgingStatsFirstDivision

Number of days to use in calculating the first claim aging bucket. This bucket includes claims between 0 and AgingStatsFirstDivision days old. This integer value must be zero (0) or greater.

Default: 30

AgingStatsSecondDivision

Number of days to use in calculating the second claim aging bucket. This bucket includes claims between AgingStatsFirstDivision + 1 and AgingStatsSecondDivision days old. This integer value must be zero (0) or greater.

Default: 60

AgingStatsThirdDivision

Number of days to use in calculating the third claim aging bucket. This bucket includes claims between `AgingStatsSecondDivision + 1` and `AgingStatsThirdDivision` days old. The last bucket includes all claims older than `AgingStatsThirdDivision` days. This integer value must be zero (0) or greater.

Default: 120

CalculateLitigatedClaimAgingStats

Whether to show the number of litigated claims on the **Aging** subtab of the **Team** tab.

Default: True

DashboardIncurredLimit

Total incurred amount above which ClaimCenter counts the claim as over-the-limit in executive dashboard calculations.

Default: 1000000

DashboardShowByCoverage

Whether the **Dashboard** shows claim information subtotalized by coverage.

Default: True

DashboardShowByLineOrLoss

Whether the **Dashboard** shows claim information subtotalized by line of business or loss type.

Default: True

DashboardWindowPeriod

Number of days to use for executive dashboard calculations that depends on a specific time period.

Default: 30

GroupSummaryShowUserGlobalWorkloadStats

Whether to show individual user global workload statistics along with the standard statistics in the **Team Summary** page.

Default: True

UserStatisticsWindowSize

Time window for calculating user statistics. Set this value to the number of previous days to include in the calculation. For example, set it to 10 to calculate statistics for the last 10 days, including today. You can also set this parameter to one of the following special values:

0	This week, defined as the start of the current business week up to and including today.
-1	This month, defined as the start of the current month up to and including today.

Default: 0

User Interface Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 34.

ActionsShortcut

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

Default: A

AutoCompleteLimit

The maximum number of autocomplete suggestions to show.

Default: 10

EnableClaimantCoverageUniquenessConstraint

If `true`, specifies that all exposure must follow the claimant and coverage uniqueness constraints.

Default: True

HidePolicyObjectsWithNoCoveragesForLossTypes

This parameter applies to policies that provide policy-level coverage rather than separate coverages for each item covered in the policy. It affects the individual coverages submenu in the **Actions** → **New Exposure** → **Choose by Coverage** submenu. For this parameter, you enter values as a comma-separated list. To remove (hide) empty Vehicle and Property submenus for a specific loss type, add that loss type to the list.

Default: None

HighlyLinkedContactThreshold

Use to improve application performance related to viewing a contact in the ClaimCenter **Address Book** tab or through the **Claim Summary** page. Attempting to view a contact with a large number of links can create performance issues. If a user is viewing a highly linked contact, then ClaimCenter issues a warning if the user clicks on a card that can result in an expensive query. The user must click another button before viewing the contact’s related claims, activities, exposures or matters as these views put a heavy load on the database. This parameter sets the threshold value for the number of links to a contact that generates the warning.

Note: If you set the threshold value to zero, then ClaimCenter considers no contact to be highly linked.

Default: None

IgnorePolicyTotalPropertiesValue

If `true`, the policy properties screens suppress the message telling the user whether all of the properties that appear on the policy have been downloaded to the ClaimCenter policy snapshot.

- Set this value to `true` if the policy adapter is not capable of returning a meaningful value for `Policy.TotalProperties`.
- Set this value to `false` otherwise.

Default: False

IgnorePolicyTotalVehiclesValue

If `true`, the policy vehicles screens suppress the message telling the user whether all of the vehicles that appear on the policy have been downloaded to the ClaimCenter policy snapshot.

- Set this value to `true` if the policy adapter is not capable of returning a meaningful value for `Policy.TotalVehicles`.
- Set this value to `false` otherwise.

Default: `False`

InputHelpTextOnFocus

If `true`, ClaimCenter displays the help text for an input any time that the input field gets the focus. (This can happen, for example, by clicking in the input field or tabbing into it.) For this to be meaningful, help text must exist. To set help text for a field, use the `helpText` attribute for that input field.

Note: Guidewire recommends that you change the value to `false` and display help text for inputs only if the mouse cursor moves over the input field.

Default: `True`

InputHelpTextOnMouseOver

If `true`, ClaimCenter displays help text for an input only if the mouse cursor moves over the input field. For this to be meaningful, help text must exist. To set help text for a field, use the `helpText` attribute for that input field.

Default: `True`

InputMaskPlaceholderCharacter

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

Default: `.` (period)

ListViewPageSizeDefault

The default number of entries that ClaimCenter 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.

MaxChooseByCoverageMenuItems

Maximum number of vehicles or properties that ClaimCenter displays in the **New Exposure → Choose by Coverage** menu. If the number to return exceeds this limit, ClaimCenter prompts the user to use the **Coverage Type** menu instead. This integer value must be one (1) or greater.

Default: `15`

MaxChooseByCoverageTypeMenuItems

Maximum number of coverage types that ClaimCenter displays in the **New Exposure → Choose by Coverage Type** menu. If the number to return exceeds this limit, ClaimCenter splits the coverage types into alphabetic submenus. This integer value must be one or greater.

Default: 15

MaxClaimantsInClaimListViews

Maximum number of claimants to list for each claim in a list view. This integer value must be zero (0) or greater. If set to zero, ClaimCenter does not impose a limit.

Default: 0

MaxTeamSummaryChartUserBars

Maximum number of users to show in the chart on the **Team Summary** page. Set this parameter to 0 to remove the chart entirely. Otherwise, the chart displays statistics for the top N users, and groups the others into a bar labeled **All Other Users**. This integer value must be zero (0) or greater.

Default: 10

QuickJumpShortcut

The keyboard shortcut to use to activate the QuickJump box.

Default: / (forward slash)

RequestReopenExplanationForTypes

The set of re-openable entities for which, if reopened, ClaimCenter displays a screen for the user to enter a reason and note. Enter as a comma-separated list.

Default: Claim,Exposure,Matter

ShowCurrentPolicyNumberInSelectPolicyDialog

Whether to populate the select policy dialog with the policy number for the current policy for a claim.

Default: False

ShowFixedExposuresInLossDetails

Works with **ShowNewExposureMenuForLossTypes**.

- If **true**, claims that do not have the **New Exposure** menu have a fixed list of exposures that can be shown through tabs on the **Claim Loss** page.
- If **false**, claims that do not have the **New Exposure** menu have a fixed list of exposures that can be shown through separate top-level page links in the claim file.

Default: True

ShowNewExposureChooseByCoverageMenuForLossTypes

Use to hide the **Actions → New Exposure → Choose By Coverage** menu for a specific loss type. In the base application configuration, the **New Exposure** menu contains two submenus, one of which is the **Choose By Coverage** submenu.

Use this parameter to hide the **Choose By Coverage** submenu for specific loss types. In general practice, Guidewire

recommends that you omit WC from this list. Enter a comma-separated list to specify the loss types for which you can create a new exposure by coverage. These values are case-sensitive.

Default: *None*

ShowNewExposureChooseByCoverageTypeMenuForLossTypes

Use to hide the **Actions** → **New Exposure** → **Choose By Coverage Type** menu for a specific loss type. In the base application configuration, the **New Exposure** contains two submenus, one of which is the **Choose By Coverage Type** submenu. Use this parameter to hide the **Choose By Coverage Type** submenu for specific loss types. In general practice, enter a comma-separated list to specify the loss types for which you can create a new exposure by coverage type. These values are case-sensitive.

Default: *None*

ShowNewExposureMenuForLossTypes

Use to hide the **Actions** → **New Exposure** menu for a specific loss type. Removing the **New Exposure** menu for a loss type also hides the **Exposures** step in the **New Claim** wizard for that loss type. Essentially, this parameter determines for which loss types you can create a new exposure. Enter a comma-separated list to specify the loss types for which the **New Exposure** menu appears. For example, enter AUTO, GL, PR to display a **New Exposures** menu for these loss types. These values are case-sensitive.

Default: *None*

UISkin

Name of the ClaimCenter 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**, ClaimCenter 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 34.

InstrumentedWorkerInfoPurgeDaysOld

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

Default: 45

WorkItemCreationBatchSize

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

Default: 100

WorkItemRetryLimit

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

Guidewire logs a `ConcurrentDataChangeException` generated by distributed workers at different levels depending on context. If the `ConcurrentDataChangeException` occurs on processing the work items, ClaimCenter logs the error only if the number of attempts exceeds the configured value of the `WorkItemRetryLimit`. Otherwise, ClaimCenter 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 core threads in the work queue thread pool. This must be greater than or equal to `WorkQueueThreadPoolMinSize`. All threads that are not core threads are additional on-demand threads, and are terminated if idle after the timeout specified by `WorkQueueThreadsKeepAliveTime`.

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 89
- “Starting Guidewire Studio” on page 90
- “The Studio Development Environment” on page 90
- “Working with the QuickStart Development Server” on page 91
- “ClaimCenter Configuration Files” on page 93

What Is Guidewire Studio?

Guidewire Studio is the ClaimCenter administration tool for creating and managing application resources. This includes Gosu rules, classes, enhancements and plugins, and all of the configuration files used by ClaimCenter in building and rendering the application.

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 ClaimCenter data entities, business objects, and data types
- Manage plugins and message destinations
- Configure database connections

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

Starting Guidewire Studio

You start Studio from the command line.

To start Guidewire Studio

1. 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:
`gwcc 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 29 for more information.

The Studio Development Environment

Guidewire Studio is a stand-alone development application that runs independently of Guidewire ClaimCenter. 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` or `.ear` 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.)

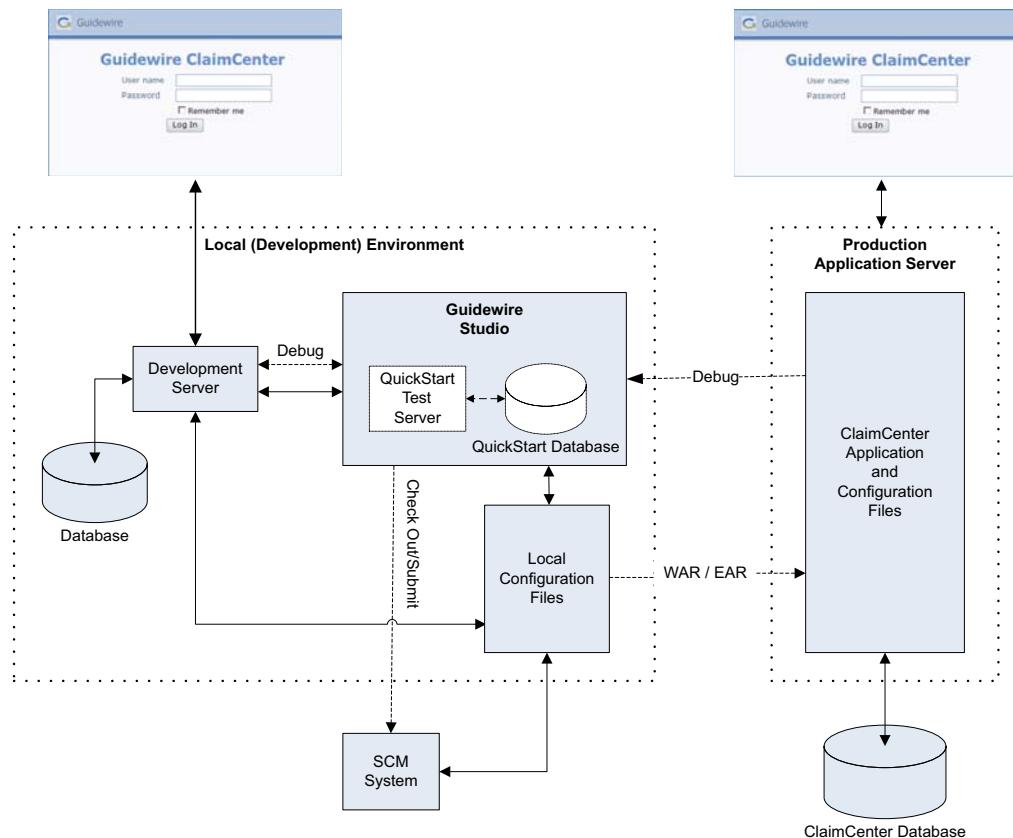
Guidewire recommends that you not run Studio on a machine with an encrypted hard drive. If you run Guidewire Studio on a machine with hard drive encryption, Studio can take 15 or more seconds to refresh. This slow refresh can happen when you switch focus from the Studio window to something else, such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

To assist with this development and testing process, Guidewire bundles the following with the ClaimCenter 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 ClaimCenter application server. You use the QuickStart test server and test

database for testing only as ClaimCenter controls them internally. You can use either the bundled QuickStart development server bundled with Guidewire ClaimCenter 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 or .ear file and manually move it to the production server. The system does not do this for you.



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 ClaimCenter to the configuration resources edited by Guidewire Studio. This requires additional configuration, described for each application server type in “Deploying ClaimCenter to the Application Server” on page 78 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

```
ClaimCenter/bin/gwcc dev-start
```

Use the following `dev` commands as you work with the QuickStart server. See “Commands Reference” on page 105 in the *Installation Guide* for a complete list of commands and how to use them.

Command	Action
<code>gwcc dev-start</code>	Starts the Development server.
<code>gwcc dev-stop</code>	Stops the Development server.
<code>gwcc 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 gwcc dev commands, see “Installing the QuickStart Development Environment” on page 46 in the *Installation Guide*.

Connecting the Development Server to a Database

ClaimCenter 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="ClaimCenterDatabase" driver="dbcp" dbtype="h2" printcommands="false"
    autoupgrade="true" checker="false">
    <param name="jdbcURL" value="jdbc:h2:file:/tmp/guidewire/cc"/>
    <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/cc"/>
```

Guidewire uses /tmp/guidewire/cc 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 ClaimCenter data model.

To drop the database, use the gwcc dev-dropdb command. To drop the database manually, delete the files from the directory specified by the jdbcURL parameter (by default, <root>/tmp/guidewire/cc). 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 or .ear 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 ClaimCenter 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.

ClaimCenter Configuration Files

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

Installing Guidewire ClaimCenter creates the following directory structure:

Directory	Description
.idea	Contains configuration and settings for IntelliJ IDEA.
admin	Contains administrative tools. See “ClaimCenter Administrative Commands” on page 181 in the <i>System Administration Guide</i> for descriptions.
bin	Contains the gwcc batch file and shell script used to launch commands for building and deploying. See “Commands Reference” on page 105 in the <i>Installation Guide</i> .
build	Contains products of build commands such as exploded .war and .ear files and the data and security dictionaries. This directory is not present when you first install ClaimCenter. The directory is created when you run one of the build commands.
dist	Guidewire application .ear, .war, and .jar files are built in this directory. The directory is created when you run one of the build commands to generate .war or .ear files.
doc	HTML and PDFs of ClaimCenter documentation.
idea	Contains IntelliJ IDEA application files.
java-api	Contains the Java API libraries created by running the gwcc regen-java-api command. See “Regenerating Integration Libraries” on page 22 in the <i>Integration Guide</i> .
logs	Contains log files.
modules	Contains subdirectories including configuration resources for each application component.
repository	Contains necessary ClaimCenter files.
soap-api	Contains the RPC-Encoded SOAP API libraries created by running the gwcc regen-soap-api command. See “Regenerating Integration Libraries” on page 22 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 ClaimCenter 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, ClaimCenter 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 ClaimCenter. In this environment, you can find all of the files needed to configure ClaimCenter in two directories:

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

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

ClaimCenter runs within a J2EE server container. To deploy ClaimCenter, 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 ClaimCenter (deployed as the `cc.war` application) running on a Tomcat J2EE server on Windows, the deployment directory might be `C:\Tomcat\webapps\cc`.

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 ClaimCenter 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 ClaimCenter 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 ClaimCenter. 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.

ClaimCenter Studio and Gosu

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

This topic includes:

- “Studio and the DCE VM” on page 97
- “Gosu Building Blocks” on page 98
- “Gosu Case Sensitivity” on page 99
- “Working with Gosu in ClaimCenter Studio” on page 100
- “Gosu Packages” on page 100
- “Gosu Classes” on page 100
- “Gosu Enhancements” on page 104
- “The Guidewire XML Model” on page 105
- “Script Parameters” on page 105

Studio and the DCE VM

The ClaimCenter 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. 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 41 in the *Installation Guide*.
- “Installing the Dynamic Code Evolution Virtual Machine” on page 42 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 ClaimCenter. 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 189 in the *Gosu Reference Guide*.
- For information on the ClaimCenter base configuration classes see, “ClaimCenter Base Configuration Classes” on page 101.
- For information on the `@export` annotation and how it affects a class in Studio, see “Class Visibility in Studio” on page 103.
- For general information on Gosu enhancements, see “Enhancements” on page 227 in the *Gosu Reference Guide*.
- For information on using Gosu business rules within Guidewire ClaimCenter, see “Rules Overview” on page 19 in the *Rules Guide*.
- For information on script parameters and how to use them in Gosu code, see “Script Parameters” on page 105.

Gosu Case Sensitivity

Gosu code compiles and runs faster if you write all your Gosu as case-sensitive code. Even though it is not currently required, Guidewire strongly recommends that you always use the proper capitalization precisely as defined for all of the following:

Language element	Standard capitalization	Example
Gosu keywords	Always specify Gosu keywords correctly as they are declared, typically lowercase. Java keywords are case-sensitive.	if
Type names	uppercase first character	DateUtil Claim
Variable names	Lowercase first character	myClaim
method names	Lowercase first character	printReport
Property names	Uppercase first character	Name
Package names (case sensitive)	Lowercase entire package name if creating new packages Always specify package names correctly as they are declared. Package names are case sensitive.	com.mycompany.*
Java types (case sensitive)	Java types require case sensitivity Always specify Java types correctly as they are declared. Java type names are case-sensitive.	java.util.String

If you do not, your code compiles slower, runs slower, and requires more memory at compile time and at run time. Additionally, using proper capitalization makes your code easier to read.

For example, if an object has a `Name` property, do not write:

```
var n = myObject.name
```

Instead, use the code:

```
var n = myObject.Name
```

Similarly, use class names properly. Do not write:

```
var a = new address()
```

Instead, use the code:

```
var a = new Address()
```

Capitalization in the middle of a word (camel-case) is also important. Do not write:

```
var date1 = gw.api.util.DateUtil.currentdate()
```

Instead, use the code:

```
var date2 = gw.api.util.DateUtil.currentDate()
```

Guidewire strongly recommends that you change any existing code to be case-sensitive code, and writing all new code to follow these guidelines. 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.

IMPORTANT Guidewire plans to strictly enforce Gosu case sensitivity in a future release. Guidewire encourages you to correct all case sensitivity issues now for performance reasons and for future upgrade compatibility.

Working with Gosu in ClaimCenter 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 189 in the <i>Gosu Reference Guide</i> “Gosu Classes” on page 100
Interface	<ul style="list-style-type: none"> “Interfaces” on page 211 in the <i>Gosu Reference Guide</i>
Enhancement	<ul style="list-style-type: none"> “Enhancements” on page 227 in the <i>Gosu Reference Guide</i> “Gosu Enhancements” on page 104
Template	<ul style="list-style-type: none"> “Gosu Templates” on page 347 in the <i>Gosu Reference Guide</i>
Package	<ul style="list-style-type: none"> “Gosu Packages” on page 100
GX Model	<ul style="list-style-type: none"> “The Guidewire XML (GX) Modeler” on page 302 in the <i>Gosu Reference Guide</i>

Gosu Packages

Guidewire ClaimCenter 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 are analogous to Java classes in that they have a package structure which contains the individual classes and the classes are extendable. Using Gosu, you can write your own custom classes and call these classes from within Gosu. You create and reference Gosu classes by name, just as you would in Java. For example, suppose that you want to define a class called `MyClass` (in package `MyPackage`) with a method called `getName`. You would first create the class (in the **Classes** folder), then call it like this:

```
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

- “ClaimCenter Base Configuration Classes” on page 101
- “Class Visibility in Studio” on page 103
- “Preloading Gosu Classes” on page 103
- “Classes” on page 189 in the *Gosu Reference Guide*

ClaimCenter 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
- libraries
- util
- wsi
- xsd

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 125.

The gw Package

In the base ClaimCenter 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 113.
- For information on various types of plugins and how to implement plugins, see “Plugin Overview” on page 163 in the *Integration Guide*.

The libraries Package

The `libraries` package contains a number of pre-built functions. To access these functions, enter the full package path (for example):

```
libraries.Activityassignment.getUserRoleGroupTypeBasedonActivityPattern( activitypattern )
```

Or, place a `uses` statement at the top of your Gosu file, which allows you to enter the library name only (for example):

```
uses libraries.Activityassignment  
...  
Activityassignment.getUserRoleGroupTypeBasedonActivityPattern( activitypattern )
```

The util Package

The `util` package contains a number of pre-built classes that provide additional functionality, including classes related to the following:

- Documents
- Financials

See also

- For information relating to documents, see “Document Creation” on page 145 in the *Rules Guide*.
- For information relating to ClaimCenter financials, see:
 - “ClaimCenter Financial Calculations” on page 619
 - “Configuring ClaimCenter Financials” on page 611
 - “Configuring ClaimCenter Financial Screens” on page 643

The wsi Package

ClaimCenter 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

- For an introduction to web services, see “Web Services Introduction” on page 31 in the *Integration Guide*.
- For the list of all built-in web services and whether each one is RPCE or WS-I, see “Reference of All Built-in Web Services” on page 33 in the *Integration Guide*.
- For background about the WS-I standard and its use of the Document Literal encoding, see “Calling WS-I Web Services from Gosu” on page 71 in the *Integration Guide*.
- For information on how to work with WS-I web services in Studio, see “Using the WS-I Web Service Editor” on page 122.

The xsd Package

As its name suggests, ClaimCenter stores XSDs in this folder. ClaimCenter uses these XSDs for a variety of functions, for example:

- `accord.xsd` – Used by ClaimCenter FNOL Mapper that maps FNOL data in the form of XML to the ClaimCenter `Claim` (and other) entity.

- `iso.xsd` – Used by ClaimCenter to generate types for ISO integration.

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

ClaimCenter 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 – ClaimCenter loads the associated Java class file. • Gosu – ClaimCenter 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 ClaimCenter 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, ClaimCenter 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 189 in the *Gosu Reference Guide*
- “**Enhancements**” on page 227 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, ClaimCenter creates an XSSD 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 302 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 105
- “**Working with Script Parameters**” on page 106
- “**Referencing a Script Parameter in Gosu**” on page 107

Script Parameters Overview

ClaimCenter 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 ClaimCenter user interface, not through Studio.

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

- **New script parameters** – ClaimCenter adds to the database new script parameters in the XML file that are not in the database. ClaimCenter propagates the initial values as set in the XML file to the database.
- **Existing script parameters** – ClaimCenter ignores existing script parameters in the XML file that already are in the database. ClaimCenter 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 ClaimCenter 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, ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 a claim has been idle for over 180 days. If you included the value “180” in all of the rules, you would have to modify the rules if you decided to change the value to 120. Instead, define a script parameter, set its value to 180, and then use this parameter in the rules. To change the claim exception behavior, you need change only the parameter and the Studio rules automatically use the new value.

More complex examples include:

- **Setting a default age for claimants** – This is useful for cases in which the computation of the reserve requires an age and none is available at the time the exposure is set up.
- **Setting the threshold number of days for various claim actions** – This includes inactive claims in which the number of inactive days before taking action varies by the coverage. After the date threshold passes, the Rule engine can generate an activity.
- **Setting the date for certain global reserve actions to occur** – Suppose that an actuary modifies the values in the reserve tables due to loss history, or to correct errors, or in response to legislation. In this case, it is possible that you want to update the reserve amounts of all open exposures. To accomplish this, you can implement several exception rules that only fire if the appropriate script parameter date is less than or equal to the current date.

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 ClaimCenter 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 ClaimCenter interface and ignores subsequent changes to values that you set as set in Studio.

IMPORTANT After you create a script parameter in Studio, ClaimCenter 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 ClaimCenter 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.parametername`.

For example, the following Gosu code determines if more than 180 days have elapsed from the filing of the claim:

```
gw.api.util.DateUtil.daysSince( Claim.ReportedDate ) > 180
```

You can, instead, create a script parameter named `maxDate` and rewrite the line as follows:

```
gw.api.util.DateUtil.daysSince(Claim.ReportedDate) > ScriptParameters.maxDate
```

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 ClaimCenter interface.

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 111
- “Working in the Gosu Editor” on page 112
- “Editing in Guidewire Studio” on page 111
- “Working in the Gosu Editor” on page 112

Editing in Guidewire Studio

Guidewire Studio displays ClaimCenter 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 143
Entity Names	Represent an entity name as a text string suitable for viewing in the ClaimCenter interface.	“Using the Entity Names Editor” on page 131
Gosu	Create and manage Gosu code used in classes, tests, enhancements, and interfaces.	“Working in the Gosu Editor” on page 112
LOB (Lines of Business)	Define the six special typelists that define the ClaimCenter Lines of Business (LOBs).	“Configuring Lines of Business” on page 477
Messaging	Work with messaging plugins.	“Using the Messaging Editor” on page 137
Page Configuration (PCF)	Graphically define and edit page configuration (PCF) files, used to render the ClaimCenter Web interface.	“Using the PCF Editor” on page 295
Plugins	Graphically define, edit and manage Java and Gosu plugins.	“Using the Plugins Registry Editor” on page 113
Typelist	Define typelists for use in the application.	“Working with Typelists” on page 271
Workflows	Graphically define and edit application workflows.	“Using the Workflow Editor” on page 387

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 189 in the *Gosu Reference Guide*



chapter 6

Using the Plugins Registry Editor

This topic covers ClaimCenter plugins. A 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 113
- “Working with Plugins” on page 114
- “Working with Plugin Versions” on page 116

What Are Plugins?

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

- An example of a plugin that calculates a result is a claim number generation plugin, which ClaimCenter invokes to generate a new claim 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 *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. Alternatively, if the ClaimCenter base configuration already provides a implementation of the plugin interface, then you can register it.

You can choose to implement a plugin as either a Gosu class, Java class, or OSGi bundle.

See “Plugin Overview” on page 163 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.

The Plugins Registry item names and fields work slightly differently depending on whether the interface supports multiple implementations. For most plugin interfaces, you can only register a single plugin implementation for that interface. However, some plugin interfaces support multiple implementations for the same interface, such as messaging plugins and startable plugins. For the maximum supported implementations for each interface, see the table “Summary of All ClaimCenter Plugins” on page 181 in the *Integration Guide*.

Each Plugin Registry item (each .gwp file) includes fields for the following core pieces of 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, ClaimCenter uses the plugin name as the interface name.

Startable Plugins

To register custom code that runs at server start up, you can write and register a startable plugin implementation, which is a plugin that implements `IStartablePlugin`. You can use a startable plugin as a listener, such as listening to a JMS queue. You can selectively start or stop each startable plugin in an administrative interface, unlike other types of plugins. Alternatively, you can use ClaimCenter multi-threaded inbound integration APIs, which use startable plugins.

Also see

- “What are Startable Plugins?” on page 271 in the *Integration Guide*
- “Multi-threaded Inbound Integration Overview” on page 279 in the *Integration Guide*

Working with Plugins

Creating a New Plugins Registry Item

You register a plugin implementation by doing the following:

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 ClaimCenter Plugins” on page 181 in the *Integration Guide*.
4. In the `Interface` box, type the name of the plugin interface, or click `Browse`  to search for valid interfaces. If you leave it blank, ClaimCenter uses the `Name` field for the interface name.

Adding a New Plugin Interface Implementation

1. In the Project window, navigate to `configuration` → `config` → `Plugins` → `registry`. In the list of plugin implementations 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 <code>EmailMessageTransport</code> plugin. See "Example Gosu Plugin" on page 169 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>ClaimCenter/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 170 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 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 115.

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 plugin 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 ClaimCenter 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 180 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 163 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	<ul style="list-style-type: none">• “Example Gosu Plugin” on page 169 in the <i>Integration Guide</i>
Java	A Java class that does not use the OSGi standard.	<ul style="list-style-type: none">• “Special Notes For Java Plugins” on page 170 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.	<ul style="list-style-type: none">• “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 ClaimCenter 8.0.0, for example, the package includes cc800.

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 119
- “Using the RPC-Encoded Web Services Editor” on page 120
- “Using the WS-I Web Service Editor” on page 122

Web Services and Guidewire Studio

Guidewire provides support for the following types of web services:

- RPCE web services or RPC-Encoded web services
- WS-I web services, for both server (publishing) and client (consuming) web service APIs

WARNING RPCE web services are deprecated. Convert existing RPCE code to WS-I web services. For more information, see “Web Services Introduction” on page 31 in the *Integration Guide*.

Within Guidewire Studio, you handle the web service types differently:

Type	In Studio...
RPCE	This type of web service appears In the Project window under the folder configuration → config → RPC-Encoded Web Services .
WS-I	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. You create a WS-I web service in Studio by doing the following: <ul style="list-style-type: none">• Navigate to a package under the configuration → config → gsrc → wsi folder.• Right-click the package and click New → WebService Collection.

The WS-I syntax is different from the RPCE syntax for both consuming and publishing web services. The following list describes the topics to consult for more information:

WS-I

["Publishing Web Services \(WS-I\)" on page 37 in the *Integration Guide*](#)

["Calling WS-I Web Services from Gosu" on page 71 in the *Integration Guide*](#)

RPCE

["Publishing Web Services \(RPCE\)" on page 89 in the *Integration Guide*](#)

["Calling RPCE Web Services from Gosu" on page 135 in the *Integration Guide*](#)

See also

- For an introduction to web services, see “Web Services Introduction” on page 31 in the *Integration Guide*.
- For the list of all built-in web services and whether each one is RPCE or WS-I, see “Reference of All Built-in Web Services” on page 33 in the *Integration Guide*.
- For background about the WS-I standard and its use of the Document Literal encoding, see “Calling WS-I Web Services from Gosu” on page 71 in the *Integration Guide*.
- For an example of calling a RPCE web service from Gosu, see “Calling RPCE Web Service from Gosu: ICD-9 Example” on page 140 in the *Integration Guide*.

Using the RPC-Encoded Web Services Editor

For information on working with web services in Guidewire ClaimCenter, see “Web Services Introduction” on page 31 in the *Integration Guide*.

WARNING RPCE web services are deprecated. Convert existing RPCE code to WS-I web services.

For more information, see “Web Services Introduction” on page 31 in the *Integration Guide*.

Guidewire provides an RPCE Web Services editor to create and manage RPCE web services within Studio. To create a new RPC web service, click **configuration** → **config** → **RPC-Encoded Web Services** in the Studio Project window, right-click, and then click **New** → **Web Service**. This action opens the **New Web Service** dialog in which you enter the name of the web service. You use this name to call this web service from Gosu code. You can enter any legal Gosu identifier. For example, a package name is acceptable. After setting the name of the web service (and clicking **OK**), the web service dialog opens.

IMPORTANT After you set your web service name, you cannot change it. It is this name that you use to create the SOAP types within Gosu code.

The RPCE Web Services editor consists of several different areas:

Area	Description
URL and Name fields	You set the Name field as you create the web service. After you set it, you cannot change it. Click Edit to enter the URL to the 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
Configuration tabs	Each web service in the base configuration contains an initial Default configuration setting in which both the env and server parameters are set to a null value. You cannot change these parameters in the default web service configuration. You can, however, create a new, named, web service configuration and set each of these parameters separately in that configuration. See “Adding a New Configuration Environment” on page 121 for more information.
Service Definitions	If you click Lookup Services in this area, Studio reads the WSDL file and displays information about any services defined in the WSDL file. For example, Studio displays the name of the service and the URL value for the service defined in the WSDL. You can also enter an override or proxy URL if you select Enable .
WSDL information	After you enter the URL for the WSDL, Studio displays information about the WSDL at the bottom of the editor in the form of the WSDL JavaDoc.

To register your web service, you must stop and restart Studio.

Adding a New Configuration Environment

Each RPCE web service in the base configuration contains an initial default setting that sets a **null** value for both the **env** and **server** parameters. You cannot change these parameters in the default plugin configuration. You can, however, create a new, named, plugin configuration and set each of these parameters separately in that configuration.

For these parameters:

- The **Env** parameter sets the deployment environment in which this plugin is active. If you have multiple deployment environments (for example, a test environment and a development environment), it is possible to make this plugin active in one of these environments only.
- The **server** parameter sets a specific server ID. If you implement ClaimCenter in a clustered environment, it is possible to make this plugin active on a certain machine within the cluster only.

For more information on these two properties, see “Reading System Properties in Plugins” on page 180 in the *Integration Guide*.

To add a new, named, configuration environment, click **Add Settings**, and set values for the **Env** and **Server** parameters. To enter these values as text strings, select the **(New)** option, then enter your values. Using these two parameters, you can create different plugin configuration environments that point to a development, test, or a production version of a WSDL. You can use the different environments to set different URLs or different authentication, for example.

You can set the following additional fields in this web services dialog:

Field	Description
Timeout (seconds)	Time to wait (in seconds) for a response from the external system. If exceeded, Gosu Runtime returns an error in the form of a Gosu exception. A timeout of zero specifies no timeout. Therefore, the system waits infinitely for a response. (Other factors might also time out the request if it takes too long, such as the ClaimCenter web application container if it contains a timeout value.)

Field	Description
Authentication	Type of authentication to use. This can be either None or HTTP. If you choose HTTP, then you must also enter a valid user name and password.
User Name/Password	The user name and password to use with this web service. Studio displays these fields only if you choose HTTP for the authentication value. If using Windows authentication, see the following "Using Web Services with Windows Authentication" on page 122.

After setting these fields, click **Lookup Services**, which is located near the bottom of the screen. This action populates the large text area with the JavaDoc associated with this web service.

Using Web Services with Windows Authentication

If you want to authenticate a web service that uses NTLM Authentication (Windows Integrated authentication), do the following:

1. Set the web service **Authentication** parameter to HTTP.
2. For the user name, specify the domain, followed by a backslash ("\"), followed by the domain user.
3. Enter a valid password.

To illustrate, suppose that you have the following:

Windows parameter	Value
User domain	winserver
User name	soapuser
Password	abc

For this example:

1. Set the HTTP user name value to this string:
`winserver\soapuser`
This is a concatenation of the domain, a backslash ("\"), and the user name.
2. Set the password as you would usually.

Using the WS-I Web Service Editor

ClaimCenter provides a fully WS-I standard-compliant web services layer for both server (publishing) and client (consuming) web service APIs. Guidewire recommends that you use the WS-I standard for all new published web services.

There are important differences in how Studio manages the different types of web services:

RPCE	Studio displays all RPCE web services as icons underneath a single icon, located under RPC-Encoded Web Services in the configuration → config tree.
WS-I	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. Note: ClaimCenter supports both the SOAP 1.1 and SOAP 1.2 protocols. Guidewire recommends, however, that you use the SOAP 1.2 protocol as the preferred protocol.

The WS-I Collections Editor

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	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	<ul style="list-style-type: none"> • Remove Resource – Use to remove a URL from the list of web service resources.
Fetch Updates	<ul style="list-style-type: none"> • 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>IWsIWebserviceConfigurationProvider</code> interface. <p>See "Adding WS-I Configuration Options" on page 78 in the <i>Integration Guide</i> for more information.</p>
Fetched Resources	Shows the XSD and WSDL files pulled from the remote host that are associated with the listed resources.

See also

- “Calling a ClaimCenter WS-I Web Service from Java” on page 66 in the *Integration Guide*
- “Loading WS-I WSDL Directly into the File System” on page 72 in the *Integration Guide*
- “Adding WS-I Configuration Options” on page 78 in the *Integration Guide*
- “Defining a Web Service Collection” on page 123

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 ClaimCenter 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.

Note: 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.

See also

- For an introduction to web services, see “Web Services Introduction” on page 31 in the *Integration Guide*.
- For the list of all built-in web services and whether each one is RPCE or WS-I, see “Reference of All Built-in Web Services” on page 33 in the *Integration Guide*.
- For background about the WS-I standard and its use of the Document Literal encoding, see “Calling WS-I Web Services from Gosu” on page 71 in the *Integration Guide*.

Implementing QuickJump Commands

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

This topic includes:

- “What Is QuickJump?” on page 125
- “Adding a QuickJump Navigation Command” on page 126
- “Checking Permissions on QuickJump Navigation Commands” on page 128

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 ClaimCenter 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 ClaimCenter. (Therefore, to remove the QuickJump box from the ClaimCenter 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 126 for details.
StaticNavigationCommandRef	Commands that navigate to a page without accepting user-entered parameters. See “Implementing StaticNavigationCommandRef Commands” on page 128.
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 128.

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. ClaimCenter 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.
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.
Command Target	<ul style="list-style-type: none"> • StaticNavigationCommandRef • ContextualNavigationCommandRef 	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 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 128 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`.

ClaimCenter displays the command in the `QuickJump` box as part of the auto-complete list (before the user has entered the entire command). Therefore, ClaimCenter 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 128 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 127.

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.

ClaimCenter 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 127 for more information.) For instance, suppose that there is an `EntityViewCommand` called `Claim` that takes a claim number and navigates to a `Claim`. Also, suppose that there is a `ContextualNavigationCommand` called `Contacts` whose context type is `Claim`. In this case, the user can type `Claim 35-402398 Contacts` to invoke the `Contacts` command on the specified `Claim`.

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 ClaimCenter 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, ClaimCenter 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, ClaimCenter 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 131
- “Variable Table” on page 132
- “Gosu Text Editor” on page 133
- “Including Data from Subentities” on page 134
- “Entity Name Types” on page 135

Entity Names Editor

It is possible to define an entity name as text string, which you can then use in the ClaimCenter 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.

ClaimCenter 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. ClaimCenter uses these entity name definitions in generating database queries that return the limited information needed to construct the display name string. This ensures that ClaimCenter 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 ClaimCenter uses in a sort
Sort Order	Defines the order in which ClaimCenter 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 ClaimCenter labels that entity column (field) as virtual in the *ClaimCenter 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 ClaimCenter 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`. ClaimCenter 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 ClaimCenter 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.

<code>Sort Path</code>	Defines the values that ClaimCenter uses in a sort
<code>Sort Order</code>	Defines the order in which ClaimCenter sorts the <code>Sort Path</code> values

Therefore, if ClaimCenter 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 ClaimCenter 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, ClaimCenter 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 ClaimCenter 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**. ClaimCenter 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 137
- “” on page 141

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 ClaimCenter 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 311 in the *Integration Guide*
- “**Implementing Messaging Plugins**” on page 343 in the *Integration Guide*
- “**Messaging and Events**” on page 299 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 ClaimCenter 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 138 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** . ClaimCenter deletes the message environment without asking for confirmation.

ClaimCenter 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 ClaimCenter 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 345 in the <i>Integration Guide</i>
----	--

Name	The name to use for this messaging destination.
Transport Plugin	<p>The name of the MessageTransport plugin implementation that knows how to send messages for this messaging destination. A destination must define a message transport plugin that sends a Message object over a physical or abstract transport. For example, the plugin might do one of the following:</p> <ul style="list-style-type: none"> • Submit the message to a message queue • Call a remote web service API and get an immediate response that the system handled the message • Implement a proprietary protocol that is specific to a remote system <p>For more information, see the following:</p> <ul style="list-style-type: none"> • “Messaging Overview” on page 300 in the <i>Integration Guide</i> • “Implementing a Message Transport Plugin” on page 345 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 (MessageRequest) plugin to prepare or pre-process a Message object before a message is sent to the message transport. For example, the MessageRequest 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 Message object before sending it. <p>To use a message reply plugin, in this Messaging editor field, type the name of the MessageRequest 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 344 in the <i>Integration Guide</i> • “Message Destination Overview” on page 311 in the <i>Integration Guide</i>
Reply Plugin	<p>A destination can optionally define a message reply (MessageReply) plugin to asynchronously acknowledge a Message object. For instance, this plugin can implement a trigger from an external system to notify ClaimCenter that the message send succeeded or failed. To use a message reply plugin, in this Messaging editor field, type the name of the MessageReply 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 346 in the <i>Integration Guide</i> • “Message Destination Overview” on page 311 in the <i>Integration Guide</i>
Alternative Primary Entity	<p>In ClaimCenter, for each message destination, messages associated with claims are always sent ordered by claim. 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 ClaimCenter with Guidewire ContactManager. It is possible that you want to safe-order messages by contact as well as by claim. In so, add Contact to this field.</p> <p>See “Message Ordering and Multi-Threaded Sending” on page 334 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 334 in the <i>Integration Guide</i>

Field	Description
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, then 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 334 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 claim 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.
Number Sender Threads	<p>To send messages associated with a claim (<i>safe-ordered</i> messages), ClaimCenter 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. ClaimCenter 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 claim 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 334 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 349 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, ClaimCenter 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"> ClaimCenter 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 334 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, ClaimCenter 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 299 in the *Integration Guide*.

See also

- For a list of built-in events that ClaimCenter triggers, see “List of Messaging Events in ClaimCenter” on page 320 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 143
- “Creating Display Keys in a Gosu Editor” on page 144
- “Retrieving the Value of a Display Key” on page 144

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.

ClaimCenter stores each display key in a `display.properties` file. If there is no international localization, ClaimCenter stores this file in the following location:

`ClaimCenter/modules/configuration/config/locale/en_US`

However, if you do localize one or more display keys, then ClaimCenter uses additional `display.properties` files, one for each locale that you create. For more information, see “Localizing Display Keys” on page 39 in the *Globalization Guide*.

ClaimCenter 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 39 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 `{}`. ClaimCenter 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, ClaimCenter 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 ClaimCenter 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 149
- “What Can You View in the Data Dictionary?” on page 150
- “Using the Data Dictionary” on page 150

What is the Data Dictionary?

The *Data Dictionary* documents all the entities and typelists in your ClaimCenter installation. Provided that you regenerate it following any customizations to the data model, the dictionary documents both the base ClaimCenter 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 ClaimCenter. 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 *ClaimCenter Data Dictionary*, run the following command from the `ClaimCenter/bin` directory:

```
gwcc regen-dictionary
```

ClaimCenter stores the current version of the *Data Dictionary* in the following directory:

```
ClaimCenter/build/dictionary/data/
```

To view the *Data Dictionary*, open the following file:

```
ClaimCenter/build/dictionary/data/index.html
```

See also

- “Regenerating the Data Dictionary and Security Dictionary” on page 30

What Can You View in the Data Dictionary?

Note: If you use a third-party tool to edit ClaimCenter 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 `ClaimCenter/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 ClaimCenter application and the columns, typekeys, arrays and foreign keys associated with each entity. “Using the Data Dictionary” on page 150 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 ClaimCenter 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 272 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**, ClaimCenter displays a left-side pane listing all of the entities in ClaimCenter. Then, on the right-side, ClaimCenter 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 ClaimCenter 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 170.</p>

Object Attributes

An object in the ClaimCenter 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 **Transaction** entity has the attributes **Abstract**, **Editable**, **Extendable**, **Keyed**, **Loadable**, **Sourceable**, **Supertype**, 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. ClaimCenter 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. ClaimCenter deletes the entity after the operation is complete.
Versionable	The entity has a version number that increases every time the entity changes. The ClaimCenter 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 ClaimCenter 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.

ClaimCenter 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.

ClaimCenter also looks at the subtype as it decides which fields to show in the ClaimCenter 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. ClaimCenter 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. ClaimCenter does not store virtual properties in the ClaimCenter physical database.

Virtual Properties on Data Entities

The *Data Dictionary* lists certain entity properties as *virtual*. ClaimCenter does not store virtual properties in the ClaimCenter 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 ClaimCenter Data Model

The in ClaimCenter *data model* comprises the persistent data objects, called *entities*, that ClaimCenter manages in the application database.

This topic includes:

- “What is the Data Model?” on page 155
- “Overview of Data Entities” on page 157
- “Base ClaimCenter Data Objects” on page 166
- “Data Object Subelements” on page 183

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 ClaimCenter 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. ClaimCenter 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 ClaimCenter data model specifies the entities, fields, and other definitions that comprise a default installation of ClaimCenter.

For example, the ClaimCenter data model defines a `Claim` entity and several fields on it, such as `ClaimNumber`, `LossDate`, and `Description`.

ClaimCenter 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. ClaimCenter stores your files that change the data model in the following application directory:

`ClaimCenter/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 ClaimCenter installation comprises the ClaimCenter model and any data model extensions that you make.

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

Working with Dot Notation

Many places within ClaimCenter require knowledge of fields within the application data model, especially while you configure ClaimCenter. 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. ClaimCenter 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 rules on a claim for example, the claim is the root object, and you can access anything that relates to this claim. On the other hand, if you run an assignment rule for an activity, the activity is the root object. In this case, you have access to fields that relate to the activity, including the claim associated with the activity.

Guidewire applications use *dot notation* for relative references. For example, assume that your code has `Claim` as the root object. For a simple reference to a field on the claim such as `LossDate`, you simply use:

```
claim.LossDate
```

However, suppose that you want to reference a field on an entity that relates to the claim, such as the policy expiration date. You must first describe the path from the claim to the policy, then describe the path from the policy to the expiration date of the policy:

```
claim.Policy.ExpirationDate
```

Overview of Data Entities

Data entities are the high-level business objects used by ClaimCenter, such as a `Claim`, `Exposure`, or `Policy`. An entity serves as the root object for data views, rules, Gosu classes, and most other data-related areas of ClaimCenter. Guidewire defines a set of data objects in the base ClaimCenter 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 ClaimCenter data model and its metadata definition files. ClaimCenter 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
TypeLists			
.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

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 ClaimCenter 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 ClaimCenter 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 ClaimCenter Address entity type.

The Extensions Directory

The `configuration/config/extensions` directory contains your data model definitions that extend the ClaimCenter data model. ClaimCenter 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 ClaimCenter data model includes the following metadata definition files that collectively define the ClaimCenter `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 ClaimCenter `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 ClaimCenter creates merged virtual directories and the directory hierarchy in general, see “ClaimCenter Configuration Files” on page 93.

The `extensions.properties` File

In general, if you change the data model by creating custom data model extensions in directory `configuration/config/extensions`, ClaimCenter 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, ClaimCenter upgrades the database.

Sometimes you want to force ClaimCenter 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 ClaimCenter. It controls whether ClaimCenter performs a database upgrade on server startup.

Whenever ClaimCenter upgrades the database, ClaimCenter stores the value of the `version` property in the database. The next time the application server starts up, ClaimCenter 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, ClaimCenter 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 **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 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.

ClaimCenter Data Entities

ClaimCenter 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 ClaimCenter 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 ClaimCenter 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, ClaimCenter 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 ClaimCenter manages the data entity in the ClaimCenter database. For example:

- If a data entity has a type value of versionable, ClaimCenter 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`, ClaimCenter stores instances of the entity in the database forever. However, you can *retire*, or hide, specific instances so that ClaimCenter does not display them in the interface.

IMPORTANT For each data entity in the ClaimCenter data model and for each entity type that you declare, ClaimCenter 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. ClaimCenter 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>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 ClaimCenter adds an instance of a <code>retireable</code> data entity to the database, ClaimCenter never deletes the instance. Instead, ClaimCenter retires the instance. For example, if you select a <code>retireable</code> instance in a list view and then click Delete, ClaimCenter preserves the instance in the database. However, ClaimCenter 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 ClaimCenter considers the instance retired.</p> <p>ClaimCenter 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 ClaimCenter 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>

ClaimCenter Database Tables

For every entity type in the data model, ClaimCenter creates a table in the application database. For example, ClaimCenter creates a `Claim` table to store information about the `Claim` object.

In the application database, you can identify an entity or extension table by the following prefix:

<code>cc_</code>	Entity table – one for each entity in the base configuration
<code>ccx_</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 176 for more information.

Besides entity tables, ClaimCenter 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>cct_</code>	Entity shadow table
<code>cctt_</code>	Typelist shadow table

ClaimCenter 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

ClaimCenter 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:

- ClaimCenter replaces foreign keys by `PublicID` objects of type `String`.
- ClaimCenter 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 191 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 `ccst_`.

Temporary (Temp) Tables

ClaimCenter 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. ClaimCenter uses temporary tables as work tables during installation or upgrade only. ClaimCenter 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 ClaimCenter defaults to a scriptability of all.

IMPORTANT There are subtle differences in how ClaimCenter treats entities and fields marked as doesNotExist and hidden. However, these differences relate to internal ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 ClaimCenter Data Objects

All ClaimCenter 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 ClaimCenter 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: ClaimCenter 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 192.

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 187.
foreignkey	See “<foreignkey>” on page 197.
fulldescription	See “<fulldescription>” on page 200.
typekey	See “<typekey>” on page 205.

Delegate Data Objects

A Delegate data object is a reusable entity that contains an interface and a *default implementation of that interface*. This permits an entity to implement an interface while delegating the implementation of that interface to another class, that of the delegate. You often use a delegate with objects that share code. The *delegate* implements the code rather than each class duplicating the shared code. Thus, a delegate is an entity associated with an implemented interface that multiple parent entities can reuse.

Guidewire defines this object in the 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, ClaimCenter 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.

Delegates 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

These are special delegates that ClaimCenter 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="retirable">
  <implementsEntity name="Validatable"/>
  ...
</entity>
```

Also, it is not possible to explicitly implement the `EventAware` delegate. ClaimCenter automatically adds this delegate to any entity that contains an `<events>` element.

Delegates that You Cannot Implement Through Extension

Guidewire does not permit you to extend Guidewire entities in the base configuration with certain archiving-related delegates. Guidewire sets these delegates carefully in the base configuration, and you cannot change them thereafter. These delegates include the following:

- `RootInfo`
- `Extractable`
- `OverlapTable`

IMPORTANT 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>`. ClaimCenter generates an error if you attempt to do so.

See also

- For an example of how to create a delegate object, see “Creating a New Delegate Object” on page 216.
- For a discussion of working with delegates in Gosu classes, see “Using Gosu Composition” on page 215 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>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>	<i>Optional.</i> Specifies an interface for which the <i>implementers</i> of this delegate must provide an implementation. By <i>implementers</i> , Guidewire means those entities that specify the delegate using <code><implementsEntity></code> . IMPORTANT This attribute is inter-related with the <code>adapter</code> attributes of <code><implementsEntity></code> . <ul style="list-style-type: none"> • If you specify a value for the <code>requires</code> attribute, then the <i>implementers</i> 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 <i>implementers</i> 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.

<delegate> subelement	Description
column	See “<column>” on page 187.
datetimeordering	<i>Internal.</i>
foreignkey	See “<foreignkey>” on page 197.
fulldescription	See “<fulldescription>” on page 200.
implementsEntity	See “<implementsEntity>” on page 200.
implementsInterface	See “<implementsInterface>” on page 201.
index	See “<index>” on page 202.
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 205.

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. ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 226

Entity Data Objects

An `Entity` data object is the standard persistent data object that Guidewire uses to define many—if not most—of the ClaimCenter 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
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 ClaimCenter 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
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 ClaimCenter.	None
exportable	Determines whether you can transmit this entity as part of a SOAP call. This attribute applies only to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or for with staging table loading, which is often known as <i>import</i> . IMPORTANT Do not set this to true on any new entities that you create.	False
extendable	<i>Internal.</i> If true, it is possible to extend this entity.	True

<entity> attribute	Description	Default
final	<p>If true, you cannot subtype the entity. If false, you can define subtypes using this entity as the supertype.</p> <p>IMPORTANT If you define this incorrectly, ClaimCenter generates an error message upon resource verification and the application server refuses to start. ClaimCenter 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. 	True
generateInternallyIfAbsent	<i>Internal.</i> Do not use.	False
ignoreForEvents	<p>If you change (or add, or remove) an entity X that does not generate events, then ClaimCenter searches for all event-generating entity instances that specify X. If ClaimCenter 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, ClaimCenter 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 ClaimCenter 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 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	<p><i>Internal.</i> If set to true, ClaimCenter adds a lock column (<code>lockingcolumn</code>) to the table for this entity. ClaimCenter 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.</p> <p>IMPORTANT Guidewire strongly recommends that you do not use this locking mechanism.</p>	False
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, because the loader import process overwrites this table.</p>	False
platform	<p><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.</p> <p>IMPORTANT Guidewire reserves the right to remove this attribute in a future release.</p>	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 164 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 ClaimCenter stores the data for this entity. ClaimCenter automatically prefixes table names with <code>cc_base</code> entities and <code>ccx_</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 ClaimCenter uses only during installation or upgrade.</p> <p>ClaimCenter 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 157 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> , ClaimCenter 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 185.
aspect	<i>Internal.</i>
checkconstraint	<i>Internal.</i>
column	See “<column>” on page 187.
componentref	See “<componentref>” on page 192.

<entity> subelement	Description
customconsistencycheck	<i>Internal.</i>
datetimeordering	<i>Internal.</i>
dbcheckbuilder	<i>Internal.</i>
edgeForeignKey	See “<edgeForeignKey>” on page 193.
events	See “<events>” on page 196.
foreignkey	See “<foreignkey>” on page 197.
fulldescription	See “<fulldescription>” on page 200.
implementsEntity	See “<implementsEntity>” on page 200.
implementsInterface	See “<implementsInterface>” on page 201.
index	See “<index>” on page 202.
jointableconsistencycheck	<i>Internal.</i>
onetoone	See “<onetoone>” on page 203.
remove-index	See “<remove-index>” on page 204.
searchColumn	See “The <searchColumn> Subelement” on page 174
tableAugmenter	<i>Internal.</i>
typekey	See “<typekey>” on page 205.
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 ClaimCenter uses the name value instead.	None
deprecated	If true, then ClaimCenter 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 184.	false
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 ClaimCenter 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*. ClaimCenter 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 209.

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 ClaimCenter 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 185.
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 213 for details.
column	See “<column>” on page 187.
column-override	Use to override certain very specific attributes of a base data object. See “Working with Attribute Overrides” on page 213 for details.
componentref	See “<componentref>” on page 192.
description	A description of the purpose and use of the entity.
edgeForeignKey	See “<edgeForeignKey>” on page 193.
foreignkey	See “<foreignkey>” on page 197.
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 213 for details.
implementsEntity	See “<implementsEntity>” on page 200.
implementsInterface	See “<implementsInterface>” on page 201.
index	See “<index>” on page 202.
internalonlyfields	<i>Internal.</i>
onetoone	See “<onetoone>” on page 203.
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 213 for details.
remove-index	See “<remove-index>” on page 204.
searchColumn	See “The <searchColumn> Subelement” on page 174
typekey	See “<typekey>” on page 205.
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 213 for details.

NonPersistent Entity Data Objects

A NonPersistentEntity data object defines a temporary, or nonpersistent, entity that ClaimCenter creates and uses only during the time that the ClaimCenter server is running. If the server shuts down, ClaimCenter 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 259 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 an create 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 ClaimCenter.	None
<code>exportable</code>	Determines whether you can transmit this entity as part of a SOAP call. This only applies to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or with staging table loading, which is often known as <i>import</i> . IMPORTANT Do not set this to true on any new entities that you create.	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 185.
aspect	<i>Internal.</i>
column	See “<column>” on page 187.
componentref	See “<componentref>” on page 192.
edgeForeignKey	See “<edgeForeignKey>” on page 193.
foreignkey	See “<foreignkey>” on page 197.
fulldescription	See “<fulldescription>” on page 200.
implementsEntity	See “<implementsEntity>” on page 200.
implementsInterface	See “<implementsInterface>” on page 201.
onetoone	See “<onetoone>” on page 203.
typekey	See “<typekey>” on page 205.

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.

ClaimCenter does not associate a separate database table with a subtype. Instead, ClaimCenter 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, ClaimCenter 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 ClaimCenter 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`. ClaimCenter 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 ClaimCenter 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> <p style="padding-left: 40px;"><code>entity.DisplayName</code></p> <p>If you do not specify a value for the displayName attribute, then ClaimCenter 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 170.</p>	None
array	The name of the subtype entity. Use this name to access the entity in data views, rules, and other areas within ClaimCenter.	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, ClaimCenter generates an error message upon resource verification and the application server refuses to start. ClaimCenter 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. ClaimCenter 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 ClaimCenter 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 185.
aspect	<i>Internal.</i>
checkconstraint	<i>Internal.</i>
column	See "<column>" on page 187.
customconsistencycheck	<i>Internal.</i>
datetimeordering	<i>Internal.</i>
dbcheckbuilder	<i>Internal.</i>
edgeForeignKey	See "<edgeForeignKey>" on page 193.
events	See "<events>" on page 196.

<subtype> subelement	Description
foreignkey	See “<foreignkey>” on page 197.
fulldescription	See “<fulldescription>” on page 200.
implementsEntity	See “<implementsEntity>” on page 200.
implementsInterface	See “<implementsInterface>” on page 201.
index	See “<index>” on page 202.
jointableconsistencycheck	<i>Internal.</i>
onetoone	See “<onetoone>” on page 203.
searchColumn	See “The <searchColumn> Subelement” on page 174
tableAugmenter	<i>Internal.</i>
typekey	See “<typekey>” on page 205.
validatetypekeyinset	<i>Internal.</i>
validatetypekeynotinset	<i>Internal.</i>

Subtypes and Typelists

After you define a new subtype, ClaimCenter automatically adds that entity type to the associated entity typelist. This is true, even if ClaimCenter 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**. ClaimCenter automatically adds the new **InspectorExt** type to the **Contact** typelist. This is true, even though ClaimCenter marks the **Contact** typelist as **final**.

To see this change:

- In the *ClaimCenter 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 221

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 of interest to a **ListView**. A **viewEntity** can include paths from the root or primary entity to other related entities.

For example, from the **ActivityView**, you can specify a column with a **Claim.ClaimNumber** value. The **Activity** entity is the primary entity of the **ActivityView**. The **Activity** entity has a corresponding **viewEntity** called **ActivityView**.

Unlike a standard **entity**, a **viewEntity** does not have an underlying database table. ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 225 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>	Determines whether you can transmit this entity as part of a SOAP call. This only applies to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or with staging table loading, which is often known as <i>import</i> .	True
	IMPORTANT Do not set this to true on any new entities that you create.	
<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 161 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 ClaimCenter 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 ClaimCenter 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 `${column_num}` passed from the ClaimCenter interface. For example, a valid expression is: `${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, ClaimCenter 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 ClaimCenter 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 (col1 + col2).
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 224 for an example of this.</p>
viewEntityLink	Uses to access another entity through a foreign key. Typically, you use this value within the ClaimCenter 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 ClaimCenter 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 `Claim` 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 ClaimCenter deletes the array elements also if you delete the array container.	False
<code>deprecated</code>	If true, then ClaimCenter 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 184.	False
<code>desc</code>	A description of the purpose and use of the array.	None
<code>exportable</code>	Determines whether you can transmit this array as part of a SOAP call. This only applies to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or with staging table loading, which is often known as <i>import</i> . IMPORTANT Do not set this to true on any new entities that you create.	True
<code>generateCode</code>	<i>Internal.</i>	true
<code>getterScriptability</code>	See “Data Objects and Scriptability” on page 164 for information.	all
<code>ignoreforevents</code>	If you change (or add, or remove) an entity X that does not generate events, then ClaimCenter searches for all event-generating entity instances that specify X. If ClaimCenter 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, ClaimCenter 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 ClaimCenter 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

<array> attribute	Description	Default
name	Required. The name of the property corresponding to this array	None
owner	If true, this entity owns the objects in the array. <ul style="list-style-type: none"> • If you delete the owning object, then ClaimCenter deletes the array items as well. • If you update the contents of the array, then ClaimCenter 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 164 for information.	all
trackssynchstate	If true, this value denotes that ClaimCenter 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 	False
	See the discussion on this attribute that follows this table.	

If set to true, the triggersValidation attribute can trigger additional ClaimCenter 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 ClaimCenter 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 ClaimCenter 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, ClaimCenter invokes the `IPreUpdateHandler` plugin on the commit bundle instead. Then, ClaimCenter executes the logic defined in the plugin on the commit bundle.
- If the parent entity has validation rules, but no preupdate rules, then ClaimCenter 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`, ClaimCenter does nothing.
 - b. In the case of `UseOldStylePreUpdate=false`, ClaimCenter calls the `IPreUpdateHandler` plugin on the commit bundle.
- In any case, any ClaimCenter processing of the commit bundle excludes the Closed and Reopened validation rules.

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 235 for more information.</p>
fulldescription	See “<fulldescription>” on page 200.
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 233 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 213 for details.

Attributes of <column>

The <column> element contains the following attributes:

<column> attribute	Description	Default
columnName	<p><i>Optional.</i> If specified, ClaimCenter uses this value as the column name of the corresponding database column. If you do not specify a columnName value, then ClaimCenter 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 213 • “Configuring Database Statistics” on page 44 in the <i>System Administration Guide</i> • “Maintenance Tools Command” on page 183 in the <i>System Administration Guide</i> 	
default	Default value given to the field during new entity creation.	None
deprecated	<p>If true, then ClaimCenter 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 184.</p>	False
desc	A description of the purpose and use of the field.	None
exportable	<p>Determines whether you can transmit this field, or column, as part of a SOAP call. This only applies to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or with staging table loading, which is often known as <i>import</i>.</p> <p>IMPORTANT Do not set this to true on any new entities that you create.</p>	True
generateCode	<i>Internal.</i>	True
getterScriptability	See “Data Objects and Scriptability” on page 164 for information.	all

<column> attribute	Description	Default
ignoreforevents	If you change (or add, or remove) an entity X that does not generate events, then ClaimCenter searches for all event-generating entity instances that specify X. If ClaimCenter 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, ClaimCenter 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 ClaimCenter 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
loadable	If true, you can load the field through staging tables. A staging table can contain a column mapping to the field.	True
name	<i>Required.</i> The name of the column on the table and the field, or property, on the entity. ClaimCenter uses this value as the column name <i>unless</i> you specify a columnName attribute. Use this name to access the column in data views, rules, and other areas within ClaimCenter. 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.	None
nullok	Whether the column can contain null values. In general, this is always true, as many tables include columns that do not require a value at different points in the process.	True
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 as the loader import process overwrites this table.	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 164 for information.	all
soapnullok	If true, then you can set the value of this column to null in SOAP calls. If you do not set this value, it defaults to the value of nullok.	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, ClaimCenter 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/cc/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 ClaimCenter 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. ClaimCenter 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"> ClaimCenter represents a column whose type is datetime in memory as instances of java.util.Date. ClaimCenter 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 ClaimCenter 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 257. For a list of the data types that you can modify or customize, see “Customizing Base Configuration Data Types” on page 261. For information on how to define new data types, see “Defining a New Data Type: Required Steps” on page 265. 	None

Subelements of <column>

The <column> element contains the following subelements:

<column> subelement	Description	Default
columnParam	See “<columnParam> Subelement” on page 190.	None
fulldescription	See “<fulldescription>” on page 200.	None
localization	See “<localization> Subelement” on page 192.	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/cc/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 ClaimCenter 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 ClaimCenter 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 <code>precision</code> 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 <code>scale</code> 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 ClaimCenter 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 253.

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 215 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 49 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 ClaimCenter 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 184.	False
<code>desc</code>	A description of the purpose and use of the array.	None
<code>exportable</code>	Determines whether you can transmit this entity as part of a SOAP call. This only applies to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or with staging table loading, which is often known as <i>import</i> . IMPORTANT Do not set this to true on any new entities that you create.	True

<componentref> attribute	Description	Default
flatten	Whether to flatten the <component> by exposing a single property that represents the <component> or inflate the <component> by exposing the individual fields of the component. For example, an entity includes a component called MoneyComponent. If the flatten attribute is true, the entity has a single property that returns the MoneyComponent. If the flatten attribute is false, the entity exposes the Amount and Currency properties of the MoneyComponent as if they were properties of the entity itself.	False
generateCode	<i>Internal.</i>	true
getterScriptability	See “Data Objects and Scriptability” on page 164 for information.	all
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 164 for information.	all

Subelements of <componentref>

The <componentref> element contains the following subelements:

<componentref> subelement	Attributes	Description	Default
annotation	<ul style="list-style-type: none"> • name • value 	Creates a name and value pair. The subelement must contain both attributes.	None
fulldescription	None	See “<fulldescription>” on page 200.	

<edgeForeignKey>

You use the <edgeForeignKey> element in a similar manner to the <foreignkey> element, to define a reference to another entity. You use an edge foreign key, however, in place of a standard foreign key if using a foreign key breaks the *safe ordering* of committing various entities to the database. Guidewire defines this element in the data model metadata files as the <edgeForeignKey> XML subelement.

IMPORTANT An edge foreign key behaves in the same fashion as a standard foreign key in the sense that it links one entity to another. Unlike a foreign key, an edge foreign key does not correspond to an actual column in the database. An edge foreign key does not implement a real database column with a foreign key constraint. ClaimCenter labels edge foreign key elements in the Guidewire *Data Dictionary* as foreign keys. You access edge foreign keys in Gosu code in the same manner as you access foreign keys.

Edge foreign keys provide the ability to avoid cyclic references in the data model. Cyclic references make an object graph that ClaimCenter cannot follow correctly. Guidewire does not permit cyclic foreign keys because there is no guarantee of a safe ordering of inserted elements into the object graph. For example, suppose that there are two entities, A and B, each of which has a foreign key to the other. Because of this relationship, it is possible that you cannot insert entity A into the object graph. This is because entity A refers to entity B, which does not yet exist in the object graph.

An edge foreign key works by creating a join array table. This table has two columns:

- OwnerID
- ForeignEntityID

If entity A has an edge foreign key to entity B, ClaimCenter creates a separate row in the edge foreign key table. In the row, `OwnerID` points to A and `ForeignEntityID` points to B.

Every time you traverse, or de-reference, the foreign key, ClaimCenter 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 ClaimCenter follows the `ForeignEntityID` on the row.

Guidewire designs edge foreign keys to work in a similar manner to standard foreign keys. You can query an edge foreign key as if it is a standard foreign key. You can also get and set edge foreign key attributes, just as you do with standard foreign keys.

Use an edge foreign key only if it is the sole way to avoid cycles in the data model that prevent a safe ordering of tables. The following are the primary cases for using an `edgeForeignKey`:

- Use if you have self-referencing foreign keys, which includes foreign keys between subtypes.
- Use if you have foreign keys from table A to table B and there is already another foreign key from table B to table A. Even more complicated cycles are possible.
- Use if you have a `Group` entity that needs to specify its parent group. This type of reference must use an edge foreign key. Otherwise, `Group` would not be safe ordered with itself.
- Use to mark the primary member of an array. You cannot use a standard foreign key. Using a foreign key would cause a cycle because the primary member must have a foreign key back to its owner.

There are more performance issues for an edge foreign key than for a standard foreign key because you must manage an entirely separate table just for that one relationship. Also, queries that require references to that column must join to an extra table. Additionally, nullability constraints do not work with edge foreign keys. You must enforce any nullability constraints by using consistency checks.

WARNING Any entity that is part of the domain graph must implement the `Extractable` delegate by including the statement `<implementsEntity name="Extractable"/>`. Otherwise, the server refuses to start. 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. The edge foreign key does not inherit the `<implementsEntity>` delegate from the enclosing entity. If you do not add it manually, the application server refuses to start.

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	If true, then ClaimCenter 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 184.	False
desc	A description of the purpose and use of the edge foreign key.	None
edgeTableEntityName	The name of the edge table entity. If you do not specify one, then ClaimCenter creates one automatically.	None
edgeTableName	<i>Required.</i> The name of the edge, or join array, table to create.	None
exportable	Determines whether you can transmit this edge foreign key as part of a SOAP call. This only applies to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or with staging table loading, which is often known as <i>import</i> . IMPORTANT Do not set this to true on any new entities that you create.	True
exportasid	If specified, ClaimCenter 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 164 for information.	all
ignoreforevents	If you change (or add, or remove) an entity X that does not generate events, then ClaimCenter searches for all event-generating entity instances that specify X. If ClaimCenter 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, ClaimCenter 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 ClaimCenter 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 ClaimCenter creates a staging table for the edge table.	False
name	Required. 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 164 for information.	all
soapnullok	If true, then you set the value of this column to null in SOAP calls. If you do not set this attribute, it defaults to the value of nullok.	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 200.
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 167.</p>

See also

- “<implementsEntity>” on page 200
- “Delegate Data Objects” on page 167

<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>
```

...

Note: ClaimCenter 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, ClaimCenter uses this value as the column name of the corresponding database column. If you do not specify a columnName value, then ClaimCenter 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 Claim, set the columnName to ClaimID.</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.	True

See "Attribute createbackingindex" on page 199 for more information.

<foreignkey> attribute	Description	Default
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>	False
	<p>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.</p>	
	See also	
	<ul style="list-style-type: none"> • “Working with Attribute Overrides” on page 213 • “Configuring Database Statistics” on page 44 in the <i>System Administration Guide</i> • “Maintenance Tools Command” on page 183 in the <i>System Administration Guide</i> 	
deprecated	<p>If true, then ClaimCenter 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>	False
	<p>If you deprecate an item, use the description to explain why.</p>	
	<p>For more information, see “The deprecated Attribute” on page 184.</p>	
desc	<p>A description of the purpose and use of the field.</p>	None
existingreferencesallowed	<p>If the following attributes are set to false, which is not the default:</p> <ul style="list-style-type: none"> • loadable • importableagainstexistingobject 	True
	<p>then, the value in the staging table can only be a reference to an existing object.</p>	
exportable	<p>Determines whether you can transmit this foreign key as part of a SOAP call. This only applies to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or with staging table loading, which is often known as <i>import</i>.</p>	True
	<p>IMPORTANT Do not set this to true on any new entities that you create.</p>	
exportasid	<p>If specified, ClaimCenter exposes the field in SOAP APIs as a string, whose value represents the PublicID of the referenced object.</p>	False
fkentity	<p>Required. The entity to which this foreign key refers.</p>	None
generateCode	<p><i>Internal.</i></p>	True
getterscriptability	<p>See “Data Objects and Scriptability” on page 164 for information.</p>	all
ignoreforevents	<p>If you change (or add, or remove) an entity X that does not generate events, then ClaimCenter searches for all event-generating entity instances that specify X. If ClaimCenter finds any of these event-generating entity instances, it generates Changed events for those entity instances.</p>	False
	<p>To determine what entities reference a non-event-generating entity, ClaimCenter 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 ClaimCenter 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. 	

<foreignkey> attribute	Description	Default
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	Required. 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 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 164 for information.	all
soapnullok	If true, then you can set the value of this column to null in SOAP calls. If you do not set this value, it defaults to the value of nullok.	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 200.

<fulldescription>

ClaimCenter 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. ClaimCenter 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.

<code><implementsEntity></code> subelement	Description
adapter	The name of the type that implements the interface specified by the <code>requires</code> attribute on <code><delegate></code> . 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.

<code><implementsInterface></code> 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 ClaimCenter to create an index on the physical database table. This index is in addition to those indexes that ClaimCenter 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, ClaimCenter frequently searches non-retired claims for one with a particular claim number. Therefore, the `Claim` entity defines an index containing both the `Retired` and `ClaimNumber` fields. However, another common search uses just `ClaimNumber`. Since that field is already part of another index, a separate index containing only `ClaimNumber` 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, ClaimCenter generates an error message upon resource verification.

Attributes of <index>

The <index> element contains the following attributes.

<index> 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 <subtype> 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, ClaimCenter 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 ClaimCenter 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 ClaimCenter 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, ClaimCenter 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: ClaimCenter 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 ClaimCenter deletes the entity to which the <onetoone> element points if you delete this entity.	False
deprecated	If true, then ClaimCenter 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 184.	False
desc	A description of the purpose and use of the field.	None

<onetoone> attribute	Description	Default
exportable	Determines whether you can transmit this entity as part of a SOAP call. This only applies to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or with staging table loading, which is often referred to as <i>import</i> .	True
	IMPORTANT Do not set this to true on any new entities that you create.	
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 164 for information.	all
ignoreforevents	If you change (or add, or remove) an entity X that does not generate events, then ClaimCenter searches for all event-generating entity instances that specify X. If ClaimCenter finds any of these event-generating entity instances, it generates Changed events for those entity instances.	False
	<p>To determine what entities reference a non-event-generating entity, ClaimCenter 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 ClaimCenter 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. 	
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	<p>If true, this entity owns the linked object (the object to which the <onetoone> element points):</p> <ul style="list-style-type: none"> If you delete the owning object, then ClaimCenter deletes the linked object as well. If you update the object pointed to by the <onetoone> element, then ClaimCenter considers the owning object updated as well. 	False
setterScriptability	See “Data Objects and Scriptability” on page 164 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 200.	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.

<remove-index> 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 271.

Attributes of <typekey>

The <typekey> element contains the following attributes.

<typekey> attribute	Description	Default
columnName	<p><i>Optional.</i> If specified, ClaimCenter uses this value as the column name of the corresponding database column. If you do not specify a columnName value, then ClaimCenter 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 213 • “Configuring Database Statistics” on page 44 in the <i>System Administration Guide</i> • “Maintenance Tools Command” on page 183 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 ClaimCenter 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 184.</p>	False
desc	A description of the purpose and use of the field.	None
exportable	<p>Determines whether you can transmit this field as part of a SOAP call. This only applies to the old-style RPC-Encoded style SOAP web services. It does not apply to WS-I web services. This attribute has nothing to do with XML import or export or with staging table loading, which is often known as <i>import</i>.</p> <p>IMPORTANT Do not set this to true on any new entities that you create.</p>	True
generateCode	<i>Internal.</i>	True
getterScriptability	See “Data Objects and Scriptability” on page 164 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 164 for information.	None
soapnullok	If true, then you can set the value of this column to null in SOAP calls. If you do not set this value, it defaults to the value of nullok.	None

<typekey> attribute	Description	Default
typefilter	The name of a filter associated with the typelist. See "Static Filters" on page 282 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 271.</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 287 for additional information.	None
fulldescription	See "<fulldescription>" on page 200.	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

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 209
- “Defining a New Data Entity” on page 212
- “Extending a Base Configuration Entity” on page 213
- “Working with Attribute Overrides” on page 213
- “Extending the Base Data Model: Examples” on page 216
- “Removing Objects from the Base Configuration Data Model” on page 225
- “Deploying Data Model Changes to the Application Server” on page 229

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 ClaimCenter 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 183.
- Create a new entity with custom fields using any of the entity types listed in “Base ClaimCenter Data Objects” on page 166.

- 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 152 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:

<code>retireable</code>	This type of entity is an extension of the <code>editable</code> entity. It is not possible to delete this entity. It is possible to retire it, however.
<code>versionable</code>	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 161 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 222 for more information.

What Happens If You Change the Data Model?

During server start up, ClaimCenter analyzes the metadata for changes since the last build. If you have made extensions, the application merges this into the working ClaimCenter data model which is the composite of the base entities and your extensions.

After merging the base data model with any extensions, ClaimCenter compares the startup layout to the physical schema in the current database. (Each ClaimCenter 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, ClaimCenter 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 ClaimCenter to report the need for an upgrade, but not actually perform it.

WARNING Do not directly modify the physical database that ClaimCenter uses. Only make changes to the ClaimCenter 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
- ClaimCenter 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

ClaimCenter 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
<array>	An array of entities
<column>	A field with a simple data type
<foreignkey>	A field referencing another entity
<typekey>	A field with a typelist

See “Data Object Subelements” on page 183 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 ClaimCenter data model. See “Deploying Data Model Changes to the Application Server” on page 229 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.

ClaimCenter 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 `Claim`, `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 ClaimCenter 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 ClaimCenter data model. See “Deploying Data Model Changes to the Application Server” on page 229 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`

tion → config → Metadata → Entity subfolders. Guidewire provides a limited number of override elements for use in .etx extension files in the configuration → config → Extensions 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 → Extensions 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
<array-override>	triggersValidation
<column-override>	createhistogram default nullok size supportsLinguisticSearch type
<foreignkey-override>	nullok triggersValidation
<onetoone-override>	triggersValidation
<typekey-override>	default nullok

These attributes have the following meanings:

Attribute	Description
createhistogram	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 maintenance_tools command. For more information on the createhistogram attribute on the column element, see “<column>” on page 187.
default	Use to change the default value given to the field (column) during new entity creation.
nullok	Use to make the nullok attribute to be more restrictive. You can only make it more restrictive, for example, changing it from nullok="true" to nullok="false".
size	Use to change the size of a column. See “A size Attribute Example” on page 215.
supportsLinguisticSearch	Use to enable linguistic search on a column.
triggersValidation	Use to determine if ClaimCenter initiates validation on changes to an array, a foreign key, or a one-to-one entity.
type	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 String column that currently is of shorttext and you want to make it use longtext. In this case, you use a <column-override> 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 190 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 ClaimCenter 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">
  <array-override name="RoleAssignments" triggersValidation="false">
  </extension>
```

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 212, you can define entirely new custom entities that become part of the ClaimCenter 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. ClaimCenter 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 ClaimCenter. 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	Regenerate the toolkit and deploy your 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.
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 229 for details. Depending on whether you are working in a development or production environment, you need to perform different tasks.

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 167 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 ClaimCenter:

- `<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 ClaimCenter, you can extend the following base configuration delegates:

- CopyOnWriteMetricLimitDelegate
- DecimalMetricDelegate
- DecimalMetricLimitDelegate
- EFTDataDelegate
- ISOMatchReport
- ISOReportable
- IntegerMetricDelegate
- IntegerMetricLimitDelegate
- MetricLimitTimeDelegate
- MoneyMetricDelegate
- MoneyMetricLimitDelegate
- PercentMetricDelegate
- PercentMetricLimitDelegate
- TimeBasedMetricDelegate
- TripAccommodationDelegate
- TripExpenseCancellationDelegate
- TripExpenseDelegate
- TripSegmentDelegate

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>`. ClaimCenter 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 216.

EFT Delegate Example

To illustrate, in the base ClaimCenter configuration, Guidewire provides a delegate named `EFTDataDelegate` to use with Electronic Funds Transfer (EFT) data.

Guidewire defines the following `EFTDataDelegate`-related files in the base ClaimCenter configuration:

File	Location	Description
<code>EFTDataDelegate.eti</code>	<code>metadata/cc</code>	Defines delegate <code>EFTDataDelegate</code> .
<code>EFTDataDelegate.etx</code>	<code>extensions</code>	Extends <code>EFTDataDelegate</code> and adds additional fields and behaviors.
<code>EFTData.eti</code>	<code>metadata/cc</code>	Defines an <code>EFTData</code> object that implements <code>EFTDataDelegate</code> .

File	Location	Description
Check.eti	metadata/cc	Creates the Check object, which implements EFTDataDelegate.
Contact.etcx	extensions	Extends the Contact entity and adds an array of EFTData objects.

The following topics describe these files.

The EFT Data Delegate

Guidewire defines the EFTDataDelegate in file EFTDataDelegate.eti. It looks similar to the following.

```
<delegate
    xmlns="http://guidewire.com/datamodel"
    extendable="true"
    javaClass="com.guidewire.cc.domain.contact.EFTDataDelegate"
    name="EFTDataDelegate">
<fulldescription>
    <![CDATA[Delegate used by Contact and Check to store Electronic funds transfer data]]>
</fulldescription>
</delegate>
```

Notice that, in this case, Java class com.guidewire.cc.domain.contact.EFTDataDelegate implements the plugin functionality. In actual practice, if you define your own delegate, you need to implement the delegate functionality in Gosu code.

Also, notice that this delegate is extendable.

The EFT Data Delegate Extension

Guidewire extends the definition of the EFTDataDelegate entity in file EFTDataDelegate.etcx.

```
<extension
    xmlns="http://guidewire.com/datamodel"
    entityName="EFTDataDelegate">
<column
    desc="The name on the account"
    name="AccountName"
    type="varchar">
    <columnParam name="size" value="100"/>
</column>
<column
    desc="The name of the bank"
    name="BankName"
    type="varchar">
    <columnParam name="size" value="100"/>
</column>
<typekey
    desc="The type of bank account e.g. checking, savings etc"
    name="BankAccountType"
    typeList="BankAccountType"/>
<column
    desc="The bank account number"
    name="BankAccountNumber"
    type="varchar">
    <columnParam name="size" value="20"/>
    <columnParam name="encryption" value="true"/>
</column>
<column
    desc="The routing number is a nine digit bank code used in the United States"
    name="BankRoutingNumber"
    type="positiveinteger"/>
<column
    desc="Indicates if this is the primary EFT record for the contact"
    name="IsPrimary"
    type="bit"/>
</extension>
```

Notice that the delegate extension adds a number of fields to the delegate that are accessible to any entity that implements this delegate, for example, AccountName and BankAccountType, among others.

EFTData Implements EFTDataDelegate

In the base ClaimCenter configuration, the EFTData entity implements the EFTDataDelegate delegate. The following abbreviated code illustrates this.

```
<!-- Entity implementing the delegate -->
<entity name="EFTData">
  <implementsEntity name=" EFTDataDelegate " />
  <foreignkey columnName="Contact"/>
</entity>
```

Check Implements EFTDataDelegate

In the base ClaimCenter configuration, the Check entity implements EFTDataDelegate directly, in a one-to-one relationship. The following abbreviated code illustrates this.

```
<!-- 1:1 relationship -->
<extension entityName="Check">
  <implementsEntity name=" EFTDataDelegate " />
</extension>
```

Contact contain array of EFTData objects

In the base ClaimCenter configuration, the Contact entity adds an array of EFTData objects. This is a one-to-many relationship. The following abbreviated code illustrates this.

```
<!-- 1:m relationship -->
<extension entityName="Contact">
  <array arrayentity="EFTData" />
</extension>
```

See also

- “The ClaimCenter Data Model” on page 155
- “Delegate Data Objects” on page 167
- “<implementsEntity>” on page 200
- “Creating a New Delegate Object” on page 216

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.

ClaimCenter 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. ClaimCenter 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 ClaimCenter bases the subtype entity). For a description of required and optional attributes, see “Subtype Data Objects” on page 178.

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 183.

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`. ClaimCenter automatically adds the new `InspectorExt` type to the `Contact` typelist. This is true, even though ClaimCenter marks the `Contact` typelist as `final`.

To see this change:

- To see this change in the *ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 ClaimCenter 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. ClaimCenter 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 157
defining a new entity	"Defining a New Data Entity" on page 212
defining an array field	"<array>" on page 185
defining a foreign key field	"<foreignkey>" on page 197

Example

The following example, defines a new retireable entity named `ExampleRetireableArrayEntityExt` and adds it as an array to the `Claim` 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 `Claim` entity. It is possible that a `Claim` 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 `Claim` extension file already exists, use CTRL-N to search for `Claim.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 `Claim.etx`.

```
<extension entityName="Claim" ...>
    ...
    <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 `Claim`:

```
<entity entity="ExampleRetireableArrayEntityExt" table="exampleretarray" ... >
    ...
    <foreignkey name="ClaimID" fkentity="Claim" desc="FK back to Claim" 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 ClaimCenter interface. (See “**viewEntity Data Objects**” on page 180 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.etc**. 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.etc**:

```
<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 <code>entity</code>	extensions	<code>.eti</code>	"Removing a Base Extension Entity" on page 226
Base configuration <code>extension</code>	extensions	<code>.etx</code>	"Removing an Extension to a Base Object" on page 227

Guidewire recommends that you review the material in “Implications of Modifying the Data Model” on page 227 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 170 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 ClaimCenter 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, ClaimCenter checks for configuration changes, such as modified extensions, that require a database upgrade. Until the database reflects the underlying configuration, ClaimCenter 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. ClaimCenter 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 226
 - “Removing an Extension to a Base Object” on page 227
2. Remove any references to the object in other parts of your configuration. If you do not remove these references, ClaimCenter 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:

```
ccx_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:

```
gwcc 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), ClaimCenter 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:

```
gwcc regen-dictionary
```

2. Create a `.war` or `.ear` file using one of the `build-*` commands:

See the “Key ClaimCenter gwcc Commands” on page 57 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), ClaimCenter runs the database upgrade tool during application start up.

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 231
- “Subtype Mapping Associative Arrays” on page 233
- “Typelist Mapping Associative Arrays” on page 235

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  
...
```

ClaimCenter uses the concept of 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 ClaimCenter 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.etc), 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 233.
- For examples of how to create a typeList associative array, see “TypeList Mapping Associative Arrays” on page 235.

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 ClaimCenter defaults to `hasGetter="true"`.

See also

- “Data Objects and Scriptability” on page 164

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 ClaimCenter *Data Dictionary*.

See also

- See “Overview of the Query Builder APIs” on page 125 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 ClaimCenter *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 334 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-associaton>` 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 334 in the *Gosu Reference Guide*.

The Domain Graph

This topic describes the ClaimCenter domain graph, what is in it, and how to access it.

This topic includes:

- “Domain Graph Overview” on page 239
- “Object Ownership and the Domain Graph” on page 240
- “Accessing the Domain Graph” on page 241
- “Adding Objects to the Domain Graph” on page 242
- “Graph Validation Tests” on page 246
- “Working with Changes to the Data Model” on page 247
- “Working with Shared Entity Data” on page 247
- “Working with Cycles” on page 248

See also

- “More Information on Archiving” on page 133 in the *Application Guide* for a list of topics related to archiving.

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

Domain Graph Overview

The domain graph is a *set of entities*. The domain graph defines an aggregate cluster of associated objects that ClaimCenter treats as a single unit for purposes of data changes. 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 Guidewire ClaimCenter, the root entity is the `Claim` object.

- The *boundary* defines what is inside the aggregate cluster of objects—all the entities that are part of the graph.

In ClaimCenter, the boundary defines the entities that relate to a `Claim` object, such as `Exposure`, `Coverage`, `Matter`, and other similar objects.

The tables associated with objects in the aggregate cluster, along with their relationships with each other, form the bounded object graph. ClaimCenter constructs the object graph by starting at the root table and collecting all the tables that are owned by the root object or its children, but excluding table views. The end result is a Directed Acyclic Graph (DAG) that starts at the root entity.

While the domain graph is central to the concept of archiving, ClaimCenter does not use it solely for archiving.

ClaimCenter uses the domain graph for the following as well:

- `Claim Purge` command line tool
- `Claim` purges called during a cancel of the `New Claim` wizard

Object Ownership and the Domain Graph

The relationships that the domain graph captures is that of ownership.

Ownership Through Foreign Keys

Guidewire defines ownership between two objects through the use of foreign keys. In general, a foreign key from object B to object A means that B is owned by A.

```
<foreignkey name="RootID"
            fkentity="TestGraphRoot"
            desc="The root of the graph and parent of this child"/>
```

Thus, the direction of the foreign key indicates the direction of ownership—which object owns the other object.

For example, in Guidewire ClaimCenter, `Claim` and `Matter` hold this relationship. In the base configuration, there is a foreign key on `Matter` that points to `Claim`. This foreign key indicates that `Matter` is owned by `Claim`.

The following `Matter` metadata definition illustrates this use of a foreign key.

```
<entity xmlns="http://guidewire.com/datamodel"
        desc="The set of data organized around a single lawsuit or potential lawsuit."
        entity="Matter" ... >
  ...
  <foreignkey columnName="ClaimID"
            desc="The claim associated with this legal matter."
            exportable="false"
            fkentity="Claim"
            name="Claim"
            nullok="false"/>
  ...
</entity>
```

Reverse Ownership in the Domain Graph

In some cases, a parent object has a reference to a child object instead of the reverse. The direction of ownership is the opposite to that of the foreign key. To indicate this direction of ownership, you set the `owner` attribute on the foreign key to `true`.

```
<foreignkey name="BackwardReferenceID"
            fkentity="TestGraphChildReferredByRoot"
            owner="true"/>
```

You must set this attribute with extension entities as well if you want to indicate this type of relationship.

Note: Guidewire actively discourages the use of reverse ownership relationships. The ClaimCenter data model supports reverse 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.

For example, `Claim` has a foreign key to `ClaimWorkComp`. In this case, `Claim` owns `ClaimWorkComp`, not the reverse.

The following `Claim` definition description illustrates this example.

```
<entity xmlns="http://guidewire.com/datamodel"
    desc="Insurance claim"
    entity="Claim" ... >
...
<foreignkey columnName="ClaimWorkCompID"
    desc="Claim's worker's compensation data"
    fkentity="ClaimWorkComp"
    name="ClaimWorkComp"
    nullok="true"
    owner="true"
    triggersValidation="true"/>
...
</entity>
```

See also

- “Working with Cycles” on page 248

Accessing the Domain Graph

Guidewire uses the DOT plain text graph description language to describe the domain graph. The DOT language provides a way of describing a complex graph of relationships by using human-readable text. Specialized software can use this text to generate a visual representation of the graph. DOT graph files generally end with the `.dot` extension.

Guidewire provides access to the domain graph through the **Server Tools** pages that are accessible to those with system administration privileges. For a discussion of the server tools, see “Using the Server Tools” on page 159 in the *System Administration Guide*.

Archiving must be enabled to access the domain graph. For information on enabling archiving, see “Archiving-related Configuration Parameters” on page 578.

The **Server Tools** → **Info Pages** → **Domain Graph Info** page contains the following two tabs, technically known as cards:

- **Graph**—Provides human-readable text versions of the domain graph. Also provides a **Download** button that generates a ZIP file containing the `.dot` definition files for the domain graph.
- **Warnings**—Provides a list of graph issues that can potentially lead to errors at application start up. Guidewire strongly recommends that you review this page any time that you make a change to the data model.

To access the domain graph

1. Log into Guidewire ClaimCenter by using an administrative account.
2. Navigate to **Server Tools**.
3. Click **Info Pages** in the left-hand side menu.
4. Select **Domain Graph Info** from the drop-down list at the top of the page.

From this screen, you can:

- View the human-readable text versions of the domain graph.
- Download a ZIP file that contains the `.dot` definition files for the domain graph.
- View (from the **Warnings** tab) any warnings generated during validation of domain graph.

Viewing the Domain Graph

You can add the path for the graphics software executable that you use to generate the `.dot` files to the `PATH` environment variable. If you do so, the download `.zip` file includes a PDF version of the graphics file as well.

To view a domain graph visually, you must download and install additional software that can read .dot files and render an image from the DOT language. There are many viewers available, some open source or free, that you can use for this purpose.

One such viewer is Graphviz, which you can download from the following URL:

<http://www.graphviz.org/About.php>

To view the domain graph visually

1. Download and install software that can read .dot files and render an image from that file.
2. Navigate to **Server Tools**, click **Download**, and save the .zip file to your local machine.
3. Extract the contents of the file into a permanent directory.

If you added the path to the graphics executable to the operating system PATH environment variable, then the download file includes a generated PDF graphic version of the DOT file. You need to complete the following steps only if you want to generate an image of the domain graph in a different graphic format.

4. Open a command window and navigate to the directory into which you extracted the .dot files.

These steps assume that you downloaded Graphviz. If you downloaded a different viewer, substitute your viewer commands.

5. Enter the following at the command prompt to generate a graphic representation of the graph files:

```
dot.exe -Tpng -o<graphic_file_name.png> <DOT_file_name>
```

This Graphviz command creates a graphic file (with a .png extension) that you can open in a graphic viewer. The graphic is quite large.

Alternatively, you can generate a PDF graphic file instead by using the following command:

```
dot.exe -Tpdf -o<graphic_file_name.pdf> <DOT_file_name>
```

Adding Objects to the Domain Graph

For ClaimCenter 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 the correct foreign keys set up to other objects both within and outside the graph.

The following table summarizes the object types and the delegates that each object type must implement.

Object	Implements...
RootInfo object	For ClaimCenter, the RootInfo object is ClaimInfo. 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.

Object	Implements...
Reference objects	A reference object is a data object that multiple instances of a domain graph object all share. In ClaimCenter, multiple <code>Claim</code> 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 ClaimCenter, the <code>Address</code> table is one such table. 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 only overlap table objects—must implement the <code>OverlapTable</code> delegate.

Any new object that you define and want to add to a graph must also correctly define a foreign key to an object in that graph. This foreign key defines which object *owns* the other.

See the following sections for details

- “Implementing the Correct Delegate” on page 243
- “Defining Foreign Keys Between Objects” on page 245

Implementing the Correct Delegate

To enforce the boundaries of the domain graph, all objects participating in the archive process must implement one or more of the following delegates:

- `RootInfo` – See “The `RootInfo` Delegate” on page 243.
- `Extractable` – See “The `Extractable` Delegate” on page 244.
- `OverlapTable` – See “The `OverlapTable` Delegate” on page 244.

A `Delegate` data object is a reusable entity that contains an interface and a default implementation of that interface. This design enables 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.

Do not attempt to change the graph to which a Guidewire base entity belongs by creating an extension that implements a different delegate. If you attempt to change the delegate that a Guidewire base entity implements through an extension entity that uses `<implementsEntity>`, ClaimCenter generates an error.

See also

- For a discussion of the Guidewire data model in general, see “The ClaimCenter Data Model” on page 155.
- For a discussion of delegate objects and how to work with them, see “Delegate Data Objects” on page 167.
- For a discussion of how to use the `<implementsEntity>` element, see “`<implementsEntity>`” on page 200.

The `RootInfo` Delegate

When ClaimCenter archives an instance of the domain graph, it leaves behind, in the main ClaimCenter database, a skeleton entity instance that provides the following information:

- Sufficient information to retrieve the data.
- Sufficient information for a minimal search on archived data.

This skeleton entity—and only this skeleton entity—must implement the `RootInfo` delegate. In Guidewire ClaimCenter, this skeleton entity is the `ClaimInfo` object, the stub object for the `Claim` object, which is the root object in the ClaimCenter domain graph. You cannot change the entity that implements the `RootInfo` delegate.

`ClaimInfo` implements the `RootInfo` delegate as follows:

```
<entity xmlns="http://guidewire.com/datamodel"
    desc="Claim Information"
    entity="ClaimInfo" ... >
    <implementsEntity name="RootInfo"/>
    ...

```

The `ClaimInfo` metadata defines minimal, summary information about the claim. For example, the object contains the claim number and loss location. ClaimCenter uses this information during claim search to find archived claims.

Note: Because ClaimCenter does not archive the `ClaimInfo` table, this table has the potential to grow very large. Guidewire recommends that you not put large amounts of data, such as a BLOB, into the table if you extend it.

The Extractable Delegate

All entities in the domain graph must implement the `Extractable` delegate. The converse is also true—no entity outside the domain graph can implement the `Extractable` delegate. The use of this delegate ensures the creation of the `ArchivePartition` column that ClaimCenter uses during the archive process.

The following metadata definition of the `Claim` object shows that it implements the `Extractable` delegate:

```
<entity xmlns="http://guidewire.com/datamodel"
    desc="Insurance claim"
    entity="Claim"
    table="claim" ... >
    <implementsEntity name="Validatable"/>
    <implementsEntity name="Assignable"/>
    <implementsEntity name="Extractable"/>
    ...

```

Any entity that is part of the domain graph must implement the `Extractable` delegate by using the `<implementsEntity>` element—the entity definition must contain `<implementsEntity name="Extractable"/>`. Otherwise, the server refuses to start. In addition, 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 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.

The OverlapTable Delegate

Overlap tables are tables whose rows 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 ClaimCenter, the `Address` table is one such table. The `Address` table itself exists in both the domain graph and as reference data.

Objects that use overlap tables must implement the `OverlapTable` delegate. Implementing the `OverlapTable` delegate creates an additional `Admin` column that ClaimCenter 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 `Address` object illustrates the use of multiple delegate implementations:

```
<entity xmlns="http://guidewire.com/datamodel"
    desc="Address of a person or business."
    entity="Address" ... >
    <implementsEntity name="Extractable"/>
    <implementsEntity name="OverlapTable"/>
    ...

```

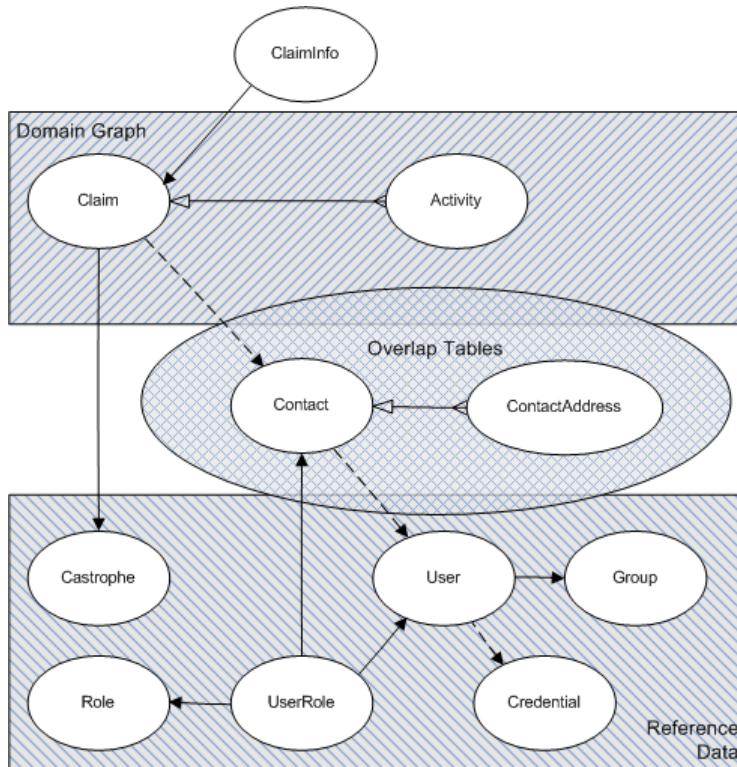
It is not common for you to have a need to use overlap tables, and thus the `OverlapTable` delegate, in your own configurations. Overlap tables are primarily for use by Guidewire internal code.

Defining Foreign Keys Between Objects

Any new object that you define and want to add to a graph must also correctly define a foreign key to an object in that graph. This foreign key defines which object *owns* the other. As discussed in “Accessing the Domain Graph” on page 241, there are several different types of ownership or ways to implement foreign keys between objects:

Ownership	Description
Object ownership	A child object (B) has a foreign key to its parent object (A). In general, a foreign key from object B to object A means that B is <i>owned by</i> A. The direction of an arrow between two objects in the graph indicates the direction of the foreign key.
Reverse ownership	A parent object (A) has a foreign key to a child object (B). Thus, the direction of ownership is opposite to that of the foreign key. To indicate this, set the <code>owner</code> attribute on the foreign key, defined on object A, to true. The graph indicates this through the use of dashed-line arrows. IMPORTANT Guidewire discourages the use of reverse ownership relationships. The ClaimCenter data model supports reverse 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.

The following illustration, which shows portions of the objects in the ClaimCenter data model, shows the relationships between objects and foreign keys. The illustration contains objects in the domain graph and in reference data, and the Contact object, which is an overlap object. The `ClaimInfo` object is outside the domain graph because it implements the `RootInfo` delegate, not the `Extractable` delegate.



In the ClaimCenter domain model:

- The `ClaimInfo` object—and only that object—implements the `RootInfo` delegate.

- All objects in the domain graph implement the `Extractable` delegate.
- The `Contact` 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.

Graph Validation Tests

The ClaimCenter server performs a series of tests on the domain graph during startup. For some of these tests, a failure prevents the server from starting. Other test failures allow the server to start with warnings. These tests warn about potential problems with the graphs that might not be an issue depending on business logic.

In ClaimCenter, these tests verify that the current data model can support archiving, purging and export of claims.

The following table describes tests that will prevent the server from starting if failed. If any of these tests fail, the server reports an exception and prints out the cycle in DOT format. You can use the DOT format graph output to view the graph using graph visualization software such as GraphViz.

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 <code>edgeTable</code> 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 cluster of associated objects, such as a Claim, its Exposures and Contacts, and so forth. ClaimCenter treats these objects as a single unit for purposes of data changes. The purging process removes non-shared data related to a Claim from the database by traversing the domain graph. Circular references in the graph cause issues for this process. 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. You need to resolve any cycles by sorting out the ownership relationships.
Domain graph entities implement <code>Extractable</code>	Ensures that all entities inside the domain graph implement the <code>Extractable</code> delegate. This test also verifies that no entities outside of the domain graph implement the <code>Extractable</code> delegate.
Overlap tables implement <code>OverlapTable</code>	Verifies that all overlap tables implement the <code>OverlapTable</code> 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"> • <code>Extractable</code> • <code>OverlapTable</code>
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. ClaimCenter has several built-in exceptions to the general rule to not have a foreign key from an entity outside the domain graph to an entity inside the domain graph. For these entities, this test verifies that the outside entity is indeed outside the domain graph.

In addition, the `Domain Graph Info` page provides warnings for situations that could potentially lead to errors. See “`Domain Graph Info`” on page 171 in the *System Administration Guide*. ClaimCenter provides warnings for these

situations rather than preventing the server from starting because business logic may prevent the erroneous situation. The following table describes these warning-level tests.

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 ClaimCenter traverses the domain graph.
Null links cannot make node unreachable	ClaimCenter constructs the domain graph by looking at foreign keys. However, this can create a <i>disconnected</i> graph if a nullable foreign key is null. If enough links are null, the graph becomes partitioned and the purging process is not able to tag the correct entities. 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.

Working with Changes to the Data Model

There are a few cases to consider if you have made changes to the data model that could potentially affect archiving:

An entity that was not in the domain graph is now in the domain graph

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 that was in the domain graph is no longer in the domain graph

This can happen, for example, because the entity was some part of reference data that was shared by multiple claims, but is now removed.

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 instance of the entity was archived already. Depending on the entity, it is possible that someone, or ClaimCenter, recreated the instance at a later time, perhaps because the entity was needed for some reason. 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 need to turn the archived instance into a referenced entity.

To handle this situation, implement the following logic:

- 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. You can create a new referenced entity of any type. The reference entity must exist in the database. If not found, then you need do nothing further.

Working with Shared Entity Data

Guidewire does not permit an entity to exist in more than one instance of the domain graph. Existence in more than one instance of the domain graph would violate the delimitation of a unit of work. However, there are cases in which you might want to share data across multiple claims.

For example, you cannot have two claims that reference the same Event entity. One way to handle this is to extend the ClaimInfo object and add a foreign key to the table for the new entity.

Extending the ClaimInfo Object

If you want to share an object, such as an Event, across multiple claims, you can extend the ClaimInfo object and add a foreign key to the 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 claims that the code references can exist in the main database.

If you intend to implement this type of solution, then Guidewire strongly recommends that you consult with Guidewire Services on the project.

Working with Cycles

There are two types of cycles that can cause issues in the ClaimCenter data model. They are:

- 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.
- 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

It is possible to have a circle in a foreign key chain. In a foreign key circle, object A has a foreign key to object B, and B has a foreign key to A. This kind of circular reference is illegal in the ClaimCenter data model. The reason is that, given a bundle containing both A and B objects, it is not possible to determine which object to commit first. It is possible that A references B, which has not yet been committed. Thus, committing A first causes a constraint violation. Any attempt to commit B first can cause the opposite problem.

To solve this kind of circular dependency between foreign keys, Guidewire recommends that you use an edge foreign key (`<edgeForeignKey>`). An edge foreign key from A to B introduces a new, hidden entity that has a foreign key to A and a foreign key to B. However, it does not create any direct foreign key from A to B. As such, ClaimCenter can safely commit the A objects first, then the B objects, and finally, the hidden A/B edge foreign key entities.

In actual practice, a circular chain of foreign keys can exist between multiple objects, not simply between two objects.

Ownership Cycles

In the domain graph, the concern is with *ownership cycles*, not simple foreign key cycles. Ownership, by default, flows in the opposite direction to a foreign key. For example, if A has a foreign key to B then B, by default, owns A.

However, it is sometimes necessary to reverse this behavior. The way to indicate this relationship is to add the special `owner="true"` attribute to the foreign key to make the ownership clear.

For example, in Guidewire ClaimCenter, the `Claim` entity has a foreign key to the `Contact` entity (`InsuredDenorm`), but `Contact` does not own `Claim`.

Note: Guidewire strongly recommends that you do not use edge foreign keys as you sort out ownership cycles in the domain graph. Do not introduce edge foreign keys into the domain graph except in the unlikely case that there is an actual safe ordering cycle that you need to correct.

See also

- “Accessing the Domain Graph” on page 241

Field Validation

This topic describes field validators in the ClaimCenter data model and how you can extend them.

This topic includes:

- “Field Validators” on page 251
- “Field Validator Definitions” on page 252
- “Modifying Field Validators” on page 255

See also

- “Configuring National Field Validation” on page 127 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 ClaimCenter 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 ClaimCenter applies the field validator uniformly to that data element in whatever code utilizes the delegate.

Field Validator Definitions

ClaimCenter stores the default field validator definitions in `fieldvalidators.xml`. This file contains a list of validator specifications for individual fields within ClaimCenter. 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 claim 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. ClaimCenter persists this value to the database, including any defined delimiters or characters other than the # character.
<code>input-mask</code>	An input-mask, which is optional, can assist the user in entering valid data. ClaimCenter 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. ClaimCenter interprets all other characters literally and includes them in the data.

After the user enters a value, ClaimCenter 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 ClaimCenter 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. ClaimCenter 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`.

ClaimCenter 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 ClaimCenter 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. ClaimCenter 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. ClaimCenter 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. ClaimCenter 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 ClaimCenter 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="ClaimNumber" value="[0-9]{3}-[0-9]{5}" description="Validator.ClaimNumber"
    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 ClaimCenter 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.

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 257
- “The Data Types Configuration File” on page 260
- “Customizing Base Configuration Data Types” on page 261
- “Working with the Medium Text Data Type (Oracle)” on page 263
- “The Data Type API” on page 263
- “Defining a New Data Type: Required Steps” on page 265
- “Defining a New Tax Identification Number Data Type” on page 265

See also

- “Monetary Amounts in the Data Model and in Gosu” on page 87 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 ClaimCenter 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 ClaimCenter 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 261.
Create a new data type	Create a .dti definition file and place it in <i>ClaimCenter/modules/configuration/config/datatypes</i> . You also need to create Gosu code to manage the data type. See “Defining a New Data Type: Required Steps” on page 265.
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="Claim">
    <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, ClaimCenter 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 ClaimCenter does not enforce this restriction at compile time. (However, ClaimCenter 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 ClaimCenter Studio) to force a database upgrade upon application server start-up.

ClaimCenter 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 ClaimCenter.

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 ClaimCenter 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 261.</p>
<code>appscale</code>	Optional attribute for use with <code>money</code> data types.
	For more information, see “The Money Data Type” on page 263.

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.

- ClaimCenter reloads the `validator` attribute for data type definitions upon server reboot. This is so that you can rebind different validators to data types.
- ClaimCenter does not reload other data type attributes such as `length`, `precision`, and `scale`. This is because ClaimCenter applies these attributes only during the initial server boot. (It uses them during table creation in

the database.) ClaimCenter ignores any changes to these attributes unless something triggers a database upgrade. For example, if you modify a base entity, then ClaimCenter 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 ClaimCenter database for the first time.
- Make modifications to the data types before performing a database upgrade that creates a new extension column.

ClaimCenter 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 262. 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 88 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}([0-9]{0,4})?"/>

```

...
 </FieldValidators>

See also

- For information on field validators in general, see “Field Validation” on page 251.
- For information on how to localize field validation, see “Configuring National Field Validation” on page 127 in the *Globalization Guide*.

List of Customizable Data Types

The following table summarizes the list of the data types that you can customize. ClaimCenter defines these data types in `datatypes.xml`. If a data type does not exist in `datatypes.xml`, then you cannot customize its attributes.

ClaimCenter 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
SSN	VARCHAR	length, validator

Guidewire data type	Built on	Customizable attributes
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 currencyamount column type in metadata definition files. For more information, see “Monetary Amounts in the Data Model and in Gosu” on page 87 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 ClaimCenter 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 ClaimCenter.

See also

- For a discussion of constraints, persistence, and presentation as it relates to data types, see “Overview of Data Types” on page 257.

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 ClaimCenter Data Model” on page 155.

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.type is 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.

Working with Typelists

This topic discusses typelists. Within Guidewire ClaimCenter, 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 ClaimCenter that presents the set of available choices. You define and manage typelists through Guidewire Studio.

This topic includes:

- “What is a Typelist?” on page 272
- “Terms Related to Typelists” on page 272
- “Typelists and Typecodes” on page 272
- “Typelist Definition Files” on page 273
- “Different Kinds of Typelists” on page 274
- “Working with Typelists in Studio” on page 275
- “Typekey Fields” on page 278
- “Removing or Retiring a Typekey” on page 280
- “Typelist Filters” on page 282
- “Static Filters” on page 282
- “Dynamic Filters” on page 287
- “Dynamic Filters” on page 287
- “Typecode References in Gosu” on page 290
- “Mapping Typecodes to External System Codes” on page 291

See also

- “Localizing Typecodes” on page 41 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 ClaimCenter data model.

Guidewire ClaimCenter 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 ClaimCenter interface, it is usually a typelist.

For example, the ClaimCenter **Loss Details** page that you access as you enter claim information contains several different typelists (drop-down lists). One of these is the **Loss Cause** typelist that provides the available values from which you can choose as you enter claim information.

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:

- `Claim.LossType` uses a simple list.
- `Claim.DriverRelationship` uses a list with a simple static filter, since only a subset of all relationships make sense in this context.
- `Claim.LossCause` uses a list filtered by `LossType` (that is, choices for this loss cause depend on the value of the loss 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 claim in ClaimCenter 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 ClaimCenter. 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 ClaimCenter.
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 ClaimCenter, 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 ClaimCenter 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 41 in the *Globalization Guide* for more information.

Mapping Typecodes to External System Codes

See the following:

- “Mapping Typecodes to External System Codes” on page 291
- “Mapping Typecodes to External System Codes” on page 143 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 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 ClaimCenter Studio. Guidewire specifically does not recommend or support manipulating the XML typelist files directly.

See also

- “Data Entity Metadata Files” on page 157

Different Kinds of Typelists

ClaimCenter organizes typelists into the following categories:

Category	Description
Internal	<p>Typelists that Guidewire controls as ClaimCenter requires these typelists for proper application operation. ClaimCenter 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 ClaimCenter to function properly. You cannot remove the internal typecodes, but you can modify any of the example typecodes.</p> <p>ClaimCenter 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

Guidewire considers a few of the typelists in the application to be internal. Guidewire controls these typelists as ClaimCenter 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 ClaimCenter 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`
- `AggregateLimitType`
- `AssignmentStatus`
- `Coverage`
- `FlaggedType`
- `PaymentType`

In some cases, Studio displays a typelist with a grayed-out icon in the **Resources** tree. This occurs if ClaimCenter 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 ClaimCenter 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. ClaimCenter 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 213.

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. ClaimCenter limits the number of characters in a typecode to 50 or less.

Working with Typelists in Studio

You create, manage and modify typelists within ClaimCenter 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 ClaimCenter.

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 modification that you make to the **PolicyType** typelist can potentially affect the **InsuranceLine** and **CoverageType** typelists that the **PolicyType** 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

ClaimCenter 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 `cctl_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 type-list in Studio. If you override the default value, the table name becomes `cctl_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 ClaimCenter 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>cctl_typeList-name</code>
Table name	Typelists editor	25 characters	<code>cctl_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 277
Filters	Define a fixed subset of a typelist to use as a static filter.	• “Static Filters” on page 282
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 287

For information on how to localize typecodes using the Studio **Typelist Localization** editor, see “Localizing Typecodes” on page 41 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. ClaimCenter 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 ClaimCenter 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 ClaimCenter data. The code must be unique within the list. ClaimCenter limits the number of characters in a typecode to 50 or less. See also “Mapping Typecodes to External System Codes” on page 291.
Name	The text that is visible within ClaimCenter 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. ClaimCenter displays the text in this field in the <i>ClaimCenter 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, ClaimCenter sorts the list alphabetically by name. If desired, you can specify priorities for some typecodes but not others. This causes ClaimCenter 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. ClaimCenter does not make changes to any existing objects that reference this typecode. If you do not enter a value, ClaimCenter 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 272
- “Mapping Typecodes to External System Codes” on page 291
- “Mapping Typecodes to External System Codes” on page 143 in the *Integration Guide*

Typekey Fields

A *typekey field* is an entity field that ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 ClaimCenter.

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 ClaimCenter Studio, through the **Typelists** editor. For information on using the **Typelists** editor, see “Working with Typelists in Studio” on page 275.

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" />
```

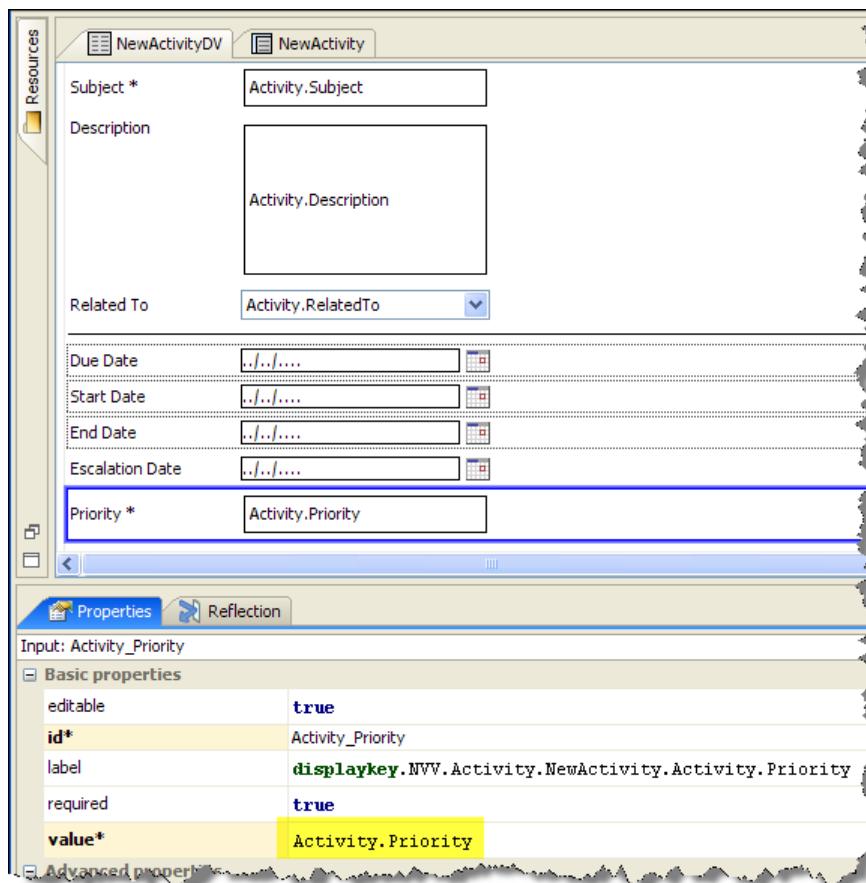
See also

- For information on the `<typekey>` element, see “`<typekey>`” on page 205.
- For information on how to create data model entities, see “The ClaimCenter Data Model” on page 155.
- For information on how to modify existing data model entities, see “Modifying the Base Data Model” on page 209.

Step 3: Reference the Typelist in the PCF File

Within Guidewire ClaimCenter, 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 ClaimCenter to render a `Priority` field on the screen, it must exist in the PCF file that ClaimCenter uses to render the screen.

Thus, the ClaimCenter **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 295.
- For information on working with PCF files in general, see “Introduction to Page Configuration” on page 309.

Step 4: Update the Product Model

Guidewire recommends that you regenerate the *ClaimCenter 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 ClaimCenter interface. Restarting the application server forces ClaimCenter 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 ClaimCenter 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. ClaimCenter 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 ClaimCenter filters the typelist values so that they do not all appear in the drop-down list (typelist) in the ClaimCenter 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 282
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 287

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>
  <typecode code="Yes" desc="Yes" name="Yes" priority="1"></typecode>
  <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 205.

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 275 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. ClaimCenter uses this value to determine if a field uses this filter.

Attribute	Description
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 ClaimCenter to use values set in the various subpanes to modify the type-list 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 283
Includes	Specific one or more typecodes to include within the filtered typelist view.	"Creating a Static Filter Using Includes" on page 285
Excludes	Specific one or more typecodes to exclude from the full list of typecodes for this typelist.	"Creating a Static Filter Using Excludes" on page 286

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 278 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 ClaimCenter application `bin` directory to regenerate the *Data Dictionary*:

```
gwcc 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 ClaimCenter user interface.
- Reference the typelist in the PCF file in which you want to display the typelist.

See “Typekey Fields” on page 278 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.

The screenshot shows the Studio Typelists editor interface. At the top, there are tabs for 'Codes' and 'Filters'. The 'Filters' tab is selected, showing a table with three rows:

Name	Description	Include All?
AustraliaFilter	Displays zone codes for	false
CAOnlyFilter	Displays unique zone cc	false
ExcludesCanada	Displays zone codes not	true

Below this is a section titled 'Includes' with a table:

Code
fsa
province

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
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.

The screenshot shows the Studio Typelist editor interface. At the top, there are two tabs: 'Codes' (selected) and 'Filters'. Below the tabs are three buttons: '+ Add', 'Duplicate', and 'Remove'. The 'Codes' pane contains a table with three rows:

Name	Description	Include All?
AustraliaFilter	Displays zone codes for	false
CAOnlyFilter	Displays unique zone c	false
ExcludesCanada1	Displays zone codes nc	true

The 'Excludes' pane below it has its own set of '+ Add', 'Duplicate', and 'Remove' buttons. It contains a table with four rows:

Code
city
fsa
province

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 ClaimCenter 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 ClaimCenter, a user with administrative privileges can define a new activity pattern (**Administration** → **Activity Patterns** → **Add Activity Pattern**). Within the **New Activity Pattern** screen, you see several drop-down lists:

- Type
- Category

ClaimCenter 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
- File Review
- General
- Interview
- ...

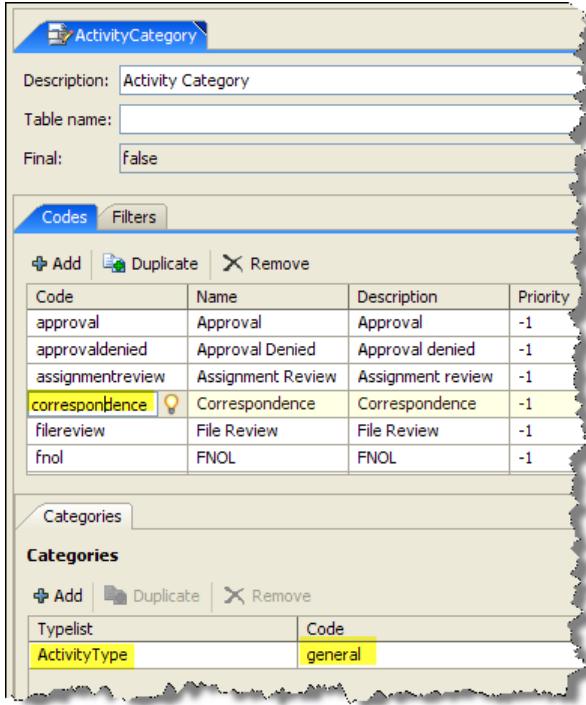
The **ActivityCategory** typelist is the typelist that controls what you see in the **Category** field in ClaimCenter. If you open this typelist in the Studio **Typelists** editor, you can choose each typecode in the list one after another. As you select each typecode in turn, notice that the Studio 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, ClaimCenter 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 **correspondence** 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.



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 ClaimCenter **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>
```

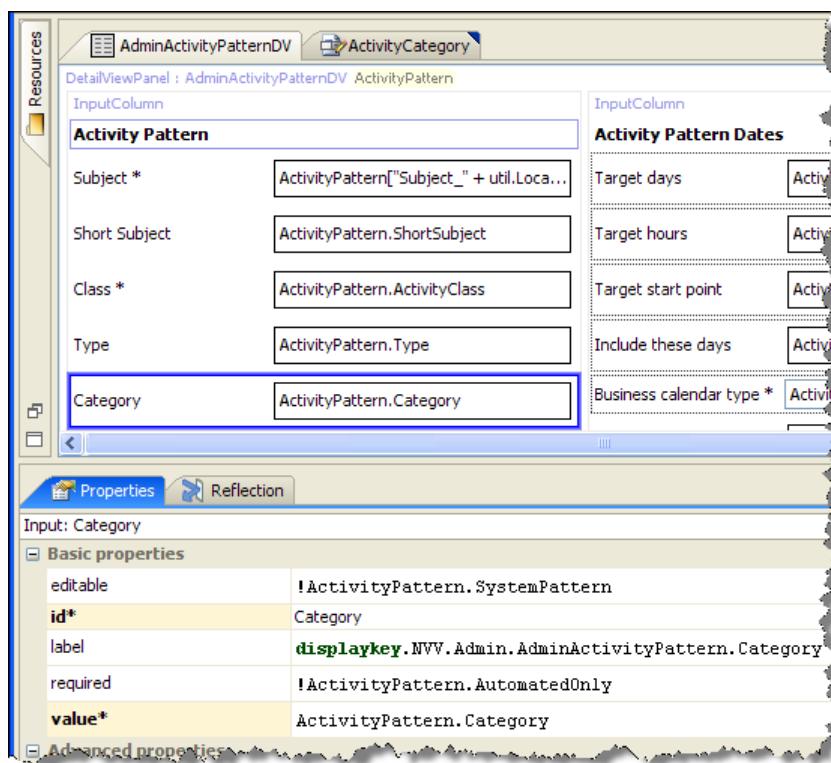
The sample code first defines a **<typekey>** element with **name="Type"** and **typelist="ActivityType"**. This is the controlling (parent) typelist. The code then defines a second typelist (**ActivityCategory**) with a keyfilter **name="Type"**. It is the typelist referenced by the **<keyfilter>** 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 205.

Step 3: Set the ClaimCenter 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 ClaimCenter **Add 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 ClaimCenter AdminActivityPatternDV.pcf file:



Step 4: Update the Product Model

Guidewire recommends that you regenerate the *ClaimCenter 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 ClaimCenter interface. Restarting the application server forces ClaimCenter 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 ClaimCenter application can share or exchange data with one or more external applications. If you use this functionality, Guidewire recommends that you configure the ClaimCenter 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 ClaimCenter typecode, but the external applications do not match. Alternatively, suppose that you extend your typecode schema in ClaimCenter. This can possibly cause a situation in which three different codes in ClaimCenter 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 more `mapping` tags. Each `mapping` tag must contain the following attributes:

<code>typecode</code>	Specifies the ClaimCenter typecode.
<code>namespace</code>	Specifies the name space to which ClaimCenter maps the typecode
<code>alias</code>	Specifies the code in the external application.

In the previous example, the PR ClaimCenter code maps to an external application named `accounting`. You can create multiple mapping entries for the same ClaimCenter typecode or the same name space. For example, the following specifies a mapping between multiple ClaimCenter 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, you use the `ITypelistToolsAPI` interface methods to translate the mappings. For more information about these methods, see the “Mapping Typecodes to External System Codes” on page 143 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 295
- “Page Canvas Overview” on page 296
- “Creating a New PCF File” on page 296
- “Working with Shared or Included Files” on page 297
- “Page Config Menu” on page 299
- “Toolbox Tab” on page 300
- “Structure Tab” on page 300
- “Properties Tab” on page 301
- “PCF Elements” on page 303
- “Working with Elements” on page 303

Page Configuration (PCF) Editor

Guidewire ClaimCenter uses *page configuration format* (PCF) files to render the ClaimCenter 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 ClaimCenter 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 ClaimCenter.

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 297 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. ClaimCenter 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, ClaimCenter 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.

ClaimCenter 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 305
<code>Edit comment...</code>	Attach a comment to any element on the canvas.	" Adding a Comment to an Element " on page 305
<code>Delete comment</code>	Remove a comment from an element.	" Adding a Comment to an Element " on page 305

Menu command	Use to...	See
Disable element	Disable an element by commenting out the widget. This prevents ClaimCenter from rendering the widget in the interface.	"Adding a Comment to an Element" on page 305
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 305
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 307
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 296
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 306
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 306

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 click `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 <code>ToolbarFlag</code> children, Studio displays the Toolbar Flags properties tab. This tab contains a child list of the <code>ToolbarFlag</code> children of the selected element.
Variables	If an element can have <code>Variable</code> children, Studio displays the Variables properties tab. This tab contains a child list of the <code>Variable</code> 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. ClaimCenter 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 ClaimCenter 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 ClaimCenter from rendering the widget. In this case, however, the XML file retains the widget. It still exists server-side at run time, although ClaimCenter 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. ClaimCenter 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 ClaimCenter user interface.

This topic includes:

- “Page Configuration Files” on page 309
- “Page Configuration Elements” on page 309
- “Getting Started Configuring Pages” on page 314
- “Modifying Style and Theme Elements” on page 316

Page Configuration Files

The pages in the ClaimCenter user interface are defined by XML files stored within each installed instance of the application. To configure your ClaimCenter 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 `ClaimCenter/modules/configuration` directory is the only place for user-edited resources. During ClaimCenter 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.

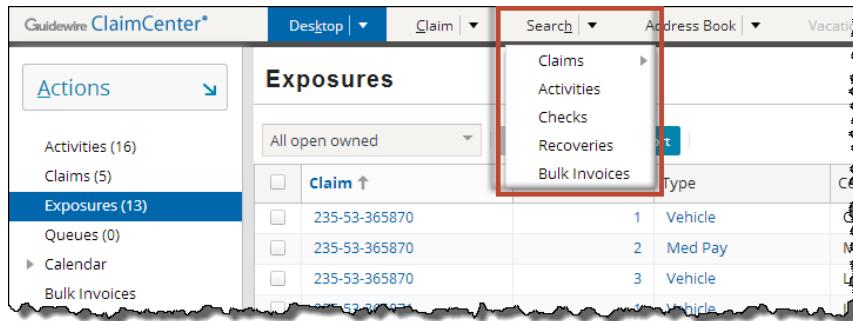
What is a PCF Element?

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 ClaimCenter interface, as well as many things that you cannot see. For example, PCF elements include:

- Editors
- List views
- Detail views
- Buttons
- Popups
- Other ClaimCenter interface elements
- Non-visible objects that support the ClaimCenter 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 `ClaimCenter/modules/pcf.html` in your installation.

Every page in ClaimCenter uses multiple PCF elements. You define these elements separately, but ClaimCenter renders them together during page construction. For example, consider the tab bar available on most ClaimCenter pages:



Using `Ctrl+Shift+W`, you can discover that these elements are defined in the `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

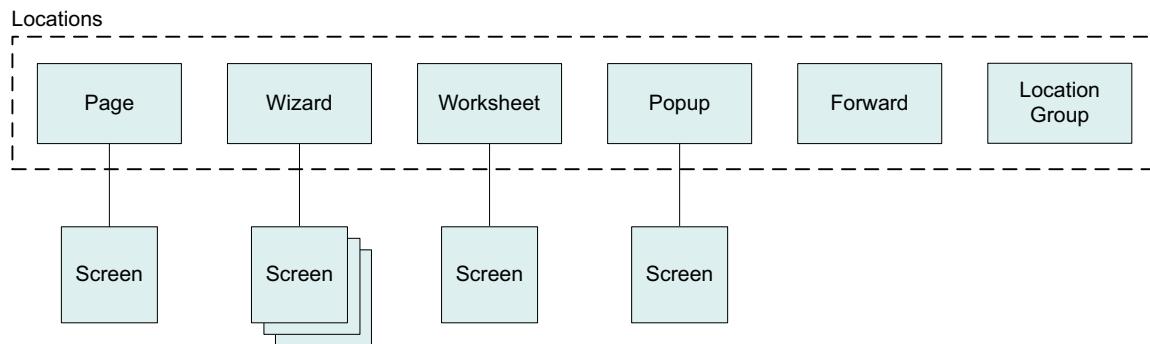
There are many kinds of PCF elements that you can define. These elements follow a hierarchical, container-based user interface model. To design them most effectively, you need understand the relationships between them thoroughly. Most PCF elements are of one of the following types:

- Locations
- Widgets

Locations

A *location* is a place to which you can navigate in the ClaimCenter 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:

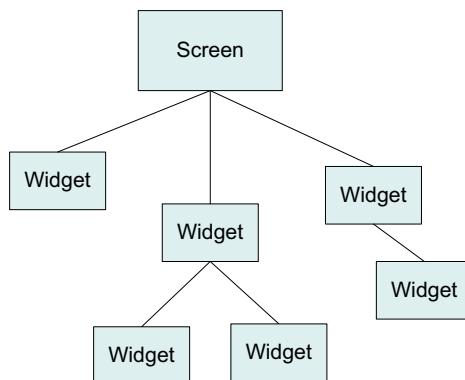
Location	Description
Page	A location with exactly one screen. The majority of locations defined in ClaimCenter are pages.
Wizard	A location with one or more screens, in which only one screen is active at a time. The contents of a wizard are usually not defined in PCF files, but are configured either in other configuration files or are defined internally by ClaimCenter.
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, ClaimCenter 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. ClaimCenter 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 ClaimCenter can render into HTML. ClaimCenter 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 ClaimCenter 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 ClaimCenter, 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:

Location (Page): *ClaimSummary*
Screen: *ClaimSummaryScreen*

Panel set: *ClaimSummaryHeadlinePanelSet*

Basics	Financials	High-Risk Indicators
Open 11 days (Target: 150) Insured hit other party's car on the front passenger side while making a left turn.	Gross Incurred \$18,400.00 Paid \$2,000.00	In litigation Currently flagged

Detail view: *ClaimSummaryDV*

Loss Date	Notice Date	Loss Location Description
08/31/2013 12:00 AM	08/31/2013	1253 Paloma Ave, Arcadia, CA 91007, United States Insured hit other party's car on the front passenger side while making a left turn.

List view: *ServiceRequestLV*

Type	Status	Service #	Next Action	Action Owner	Relates To	Services	Vendor
		1001	Approve quote	Andy Applegate	Claim	Audio equipment Auto body	Mike's Auto detect
		1002	Submit request	Andy Applegate	Claim	Auto body	Mike's Auto detect

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`.

ClaimCenter provides the following tools that allows you to view the structure of any page and to see which PCF elements it uses:

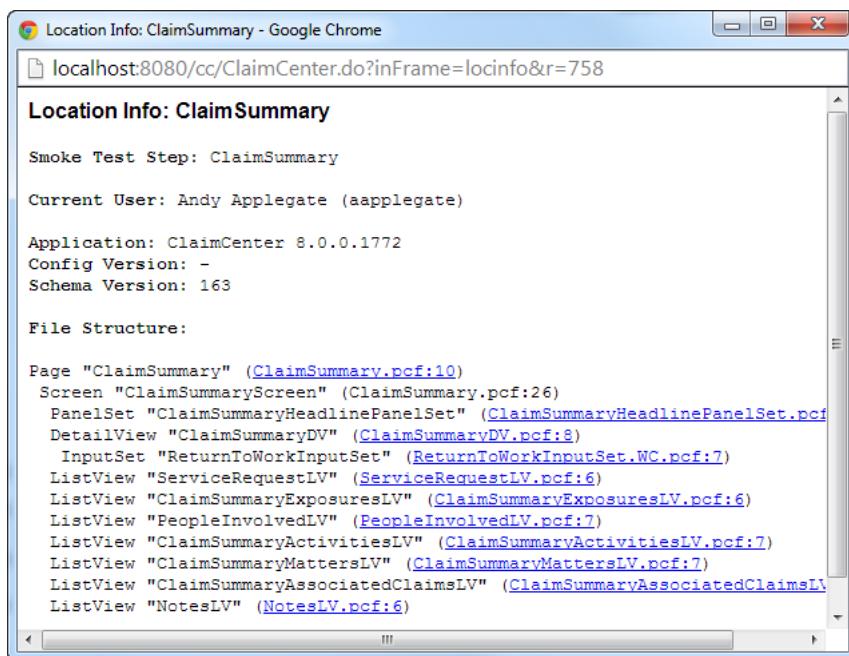
- Location Info
- Widget Inspector

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 ClaimCenter 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:



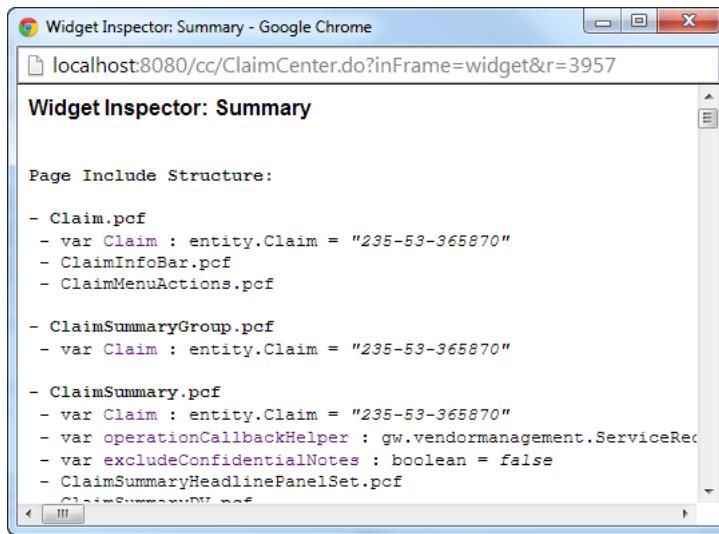
With this information, you can see:

- 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 ClaimCenter 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

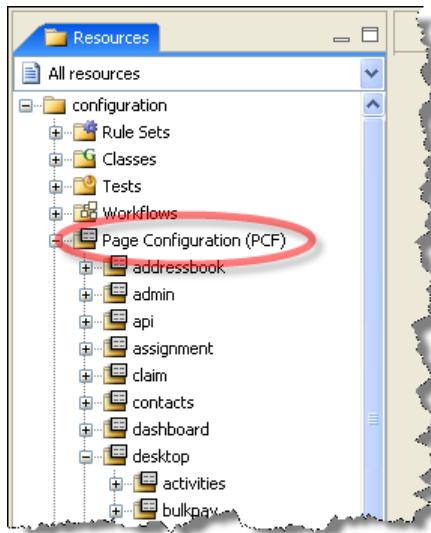
Finding an Existing Element To Edit

The first step in modifying the ClaimCenter 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

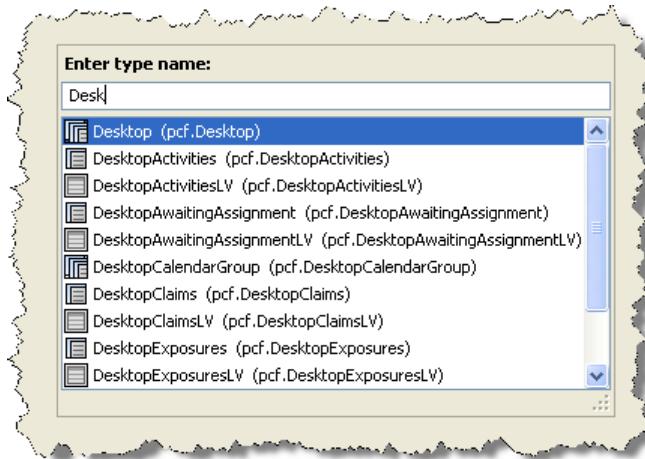
You can browse the PCF elements under the **Page Configuration** folder in Guidewire Studio:



These elements are arranged in a folder hierarchy that is related to how they appear in the ClaimCenter 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, ClaimCenter displays a list of possible elements that match the ID you are entering.



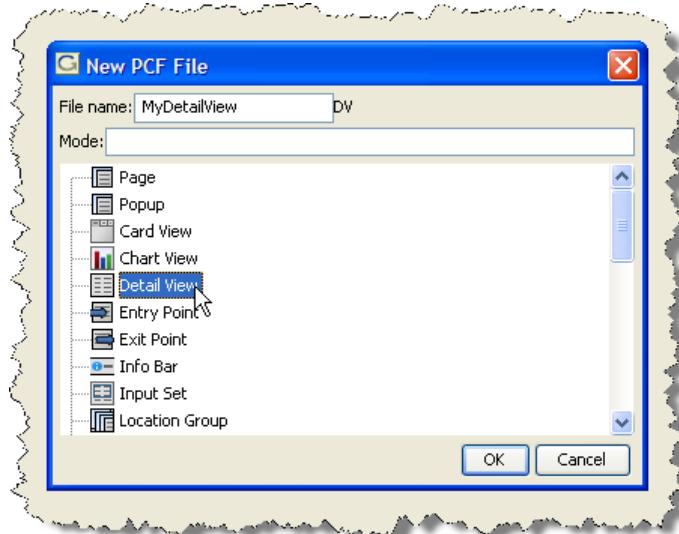
After you see the one you want, click on it and ClaimCenter opens the file in the PCF Editor.

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 `ClaimCenter/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 `gwcc update-theme` from the command line.

Overriding CSS

To override specific CSS classes, make edits to `ClaimCenter/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 317. However, you can change the theme colors of your application by following these steps:

1. Open `ClaimCenter/ThemeApp/packages/titanium/sass/var/Component.scss` in a text editor. This is where all of the ClaimCenter 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 `gwcc 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

ClaimCenter 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 `ClaimCenter/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 `gwcc dev-deploy-web-resources-debug` and refresh your browser.
 - `gwcc 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 ClaimCenter user interface.

This topic includes:

- “Panel Overview” on page 319
- “Detail View Panel” on page 319
- “List View Panel” on page 324

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:

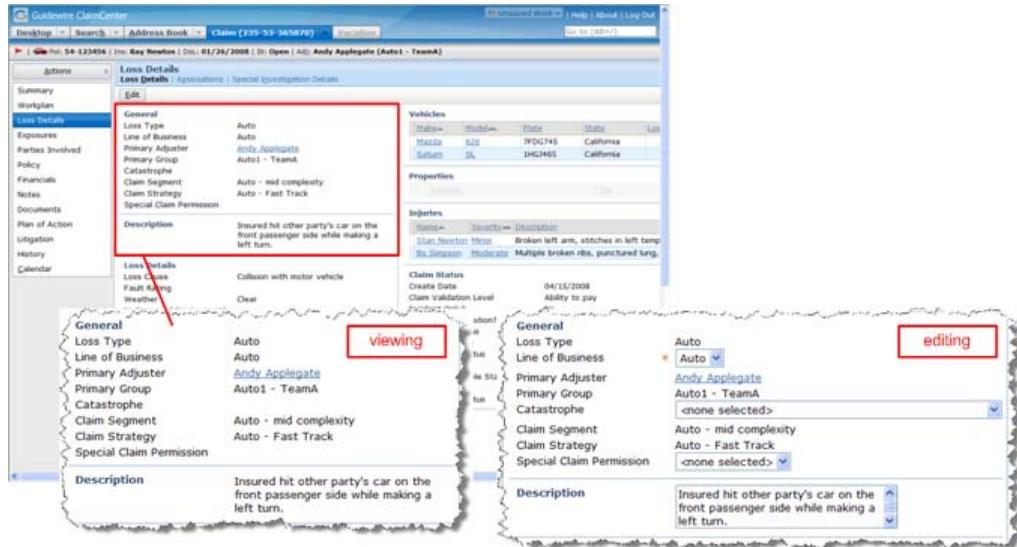
- Detail View Panel – A series of widgets laid out in one or more columns.
- List View Panel – A list of array objects, or any other data that can be laid out in tabular form.

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:

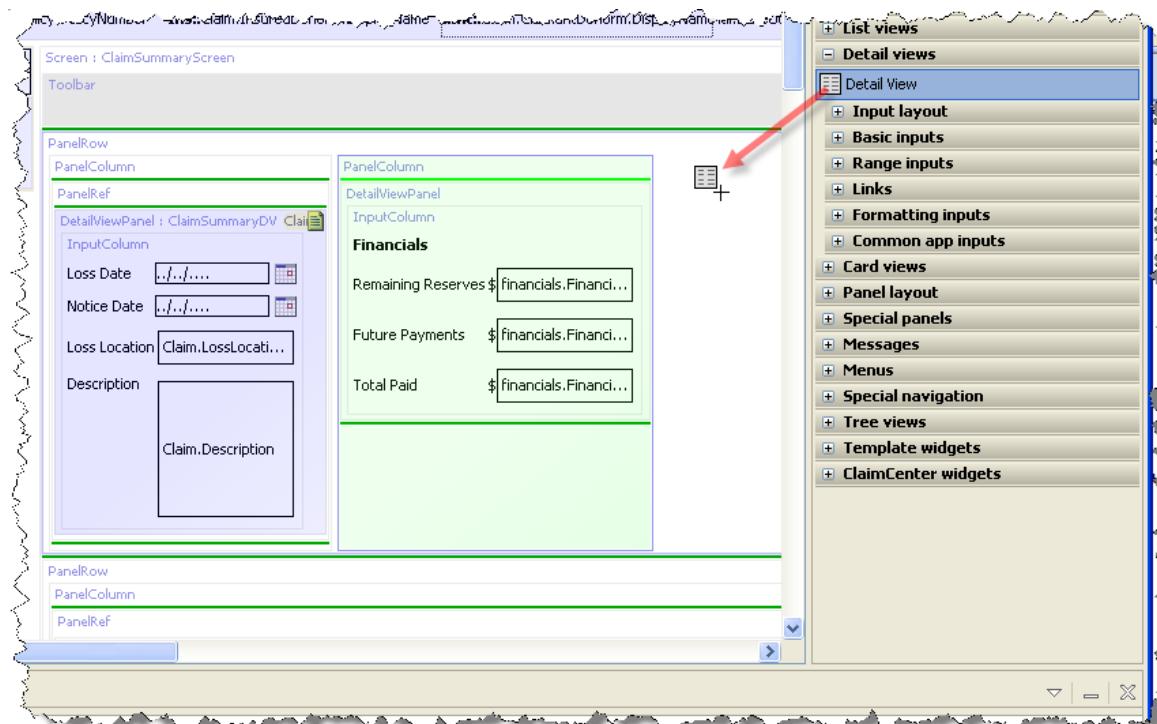


You can do the following:

- Define a Detail View
- Add Columns to a Detail View
- Format a Detail View

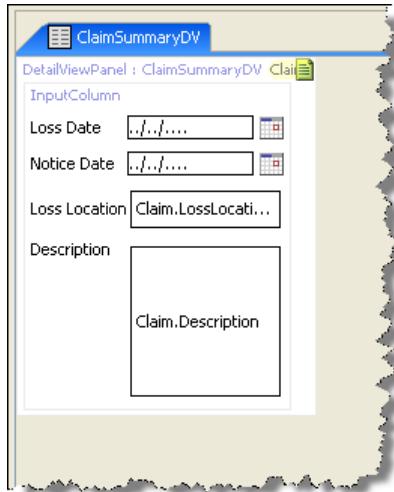
Define a Detail View

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:

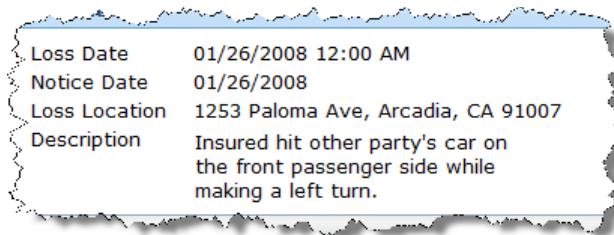


The `id` attribute is required; it identifies the panel so that it can be referenced by other PCF elements. The ID must be unique, and it must end with the text string DV.

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

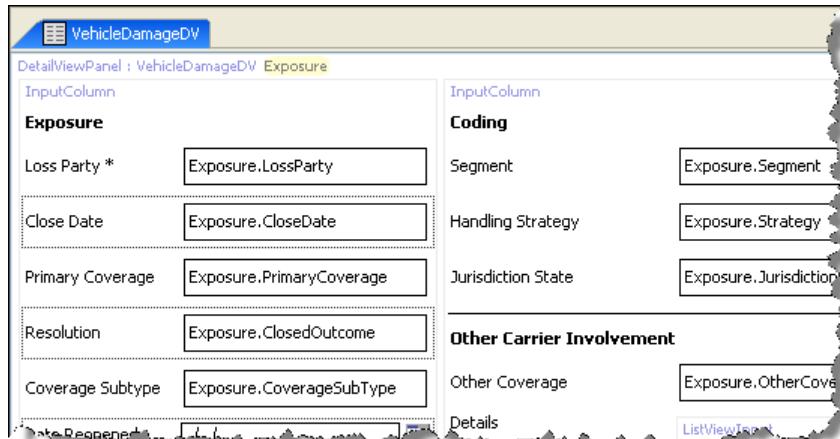
A detail view must contain at least one vertical column, but it can contain more. The following illustration shows detail views with one and two columns:

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:



ClaimCenter 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	-

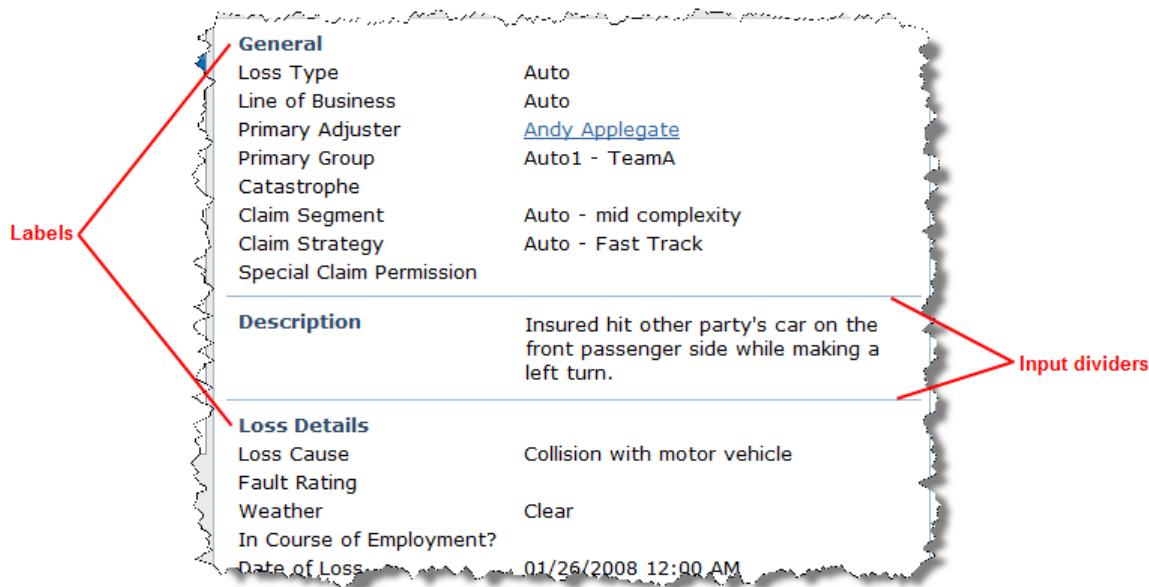
vertical divider

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:

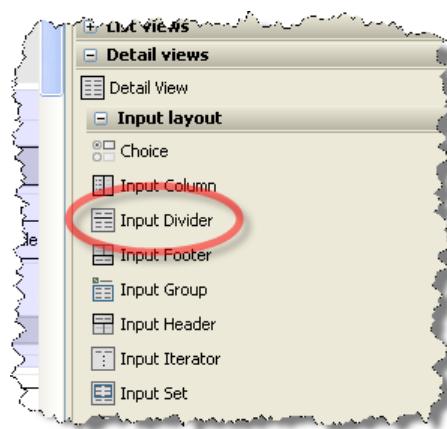


Set the `label` attribute to the display key to use for the label.

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:

The diagram illustrates a list view panel titled "Exposures". The panel includes a toolbar at the top with buttons for "Assign", "Refresh", "Close Exposure", "Create Reserve", and "Print/Export". A "filter" dropdown menu is located on the left. The main area contains a table with the following data:

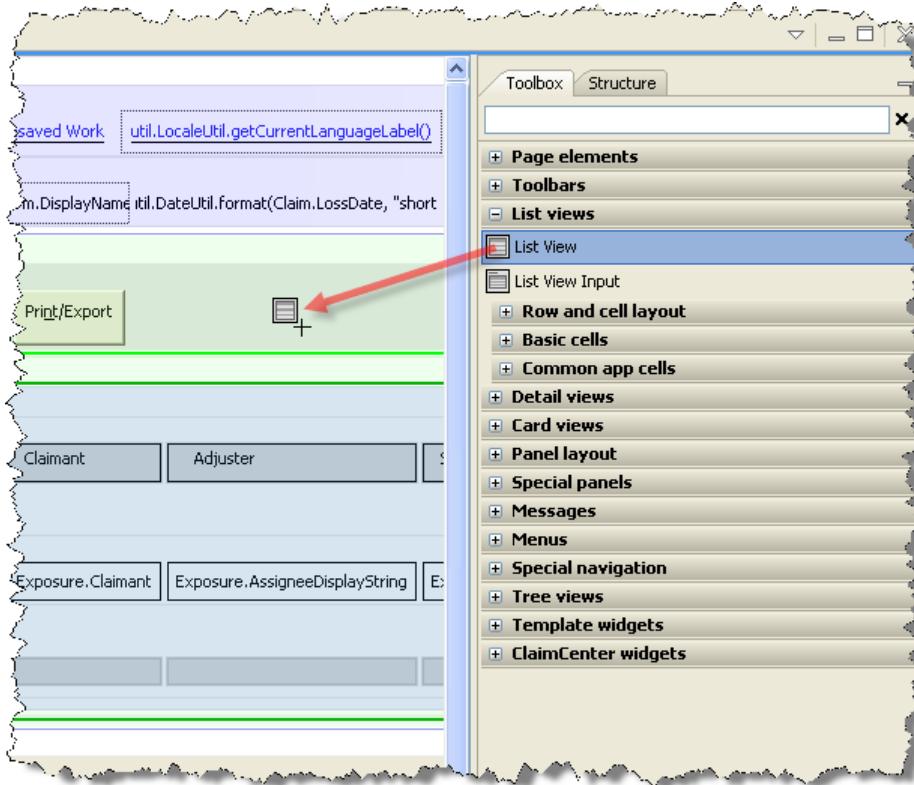
#	Type	Coverage	Claimant	Adjuster	Status	Remaining Reserves	Future Payments	Paid
1	Vehicle	Collision	Ray Newton	Andy Applegate	Open	\$400.00	-	\$500.00
2	Med Pay	Medical payments	Stan Newton	Andy Applegate	Open	\$2,000.00	-	\$1,500.00
3	Vehicle	Liability - Property damage	Bo Simpson	Andy Applegate	Open	\$5,000.00	-	-
4	Bodily Injury	Liability - Auto bodily injury	Bo Simpson	Carla Levitt	Open	\$9,000.00	-	-

Annotations with red arrows point to specific parts of the interface:

- "filter" points to the dropdown menu on the left.
- "toolbar" points to the top row of buttons.
- "header" points to the top-most row of the table.
- "rows" points to the individual data rows in the table.
- "columns" points to the column headers.

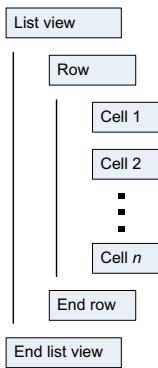
Define a List View

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:

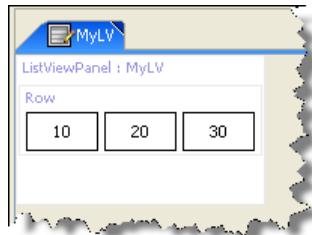


The `id` attribute is required; it identifies the panel so that it can be referenced by other PCF elements. The ID must be unique, and it must end with the text string `LV`.

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 ClaimCenter. 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 327.

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:

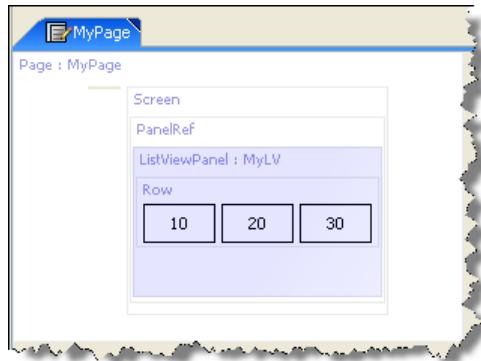
- Standalone
- Inline

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:

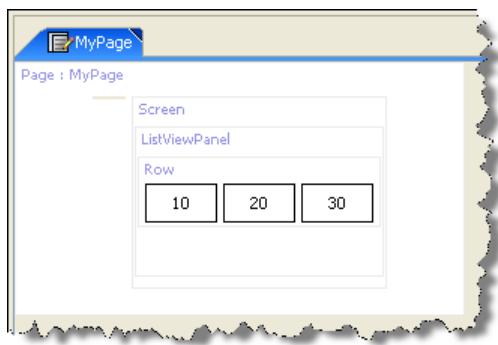


Set the `def` attribute of the `PanelRef` to the name of the list view; in this example, that is `MyLV`.

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.

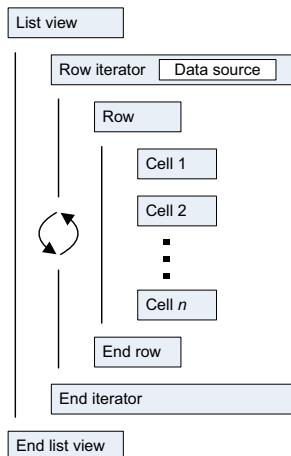
The following example defines an inline list view in a screen:



Iterate a List View Over a Data Set

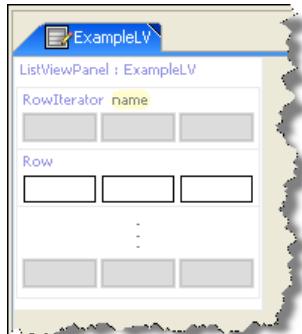
Most list views iterate over a data set and dynamically create a new row in the list for each record in the data set. The most common usage is showing an array of objects that belong to another object. For example, listing all activities that belong to a claim, or all users that belong to a group.

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.

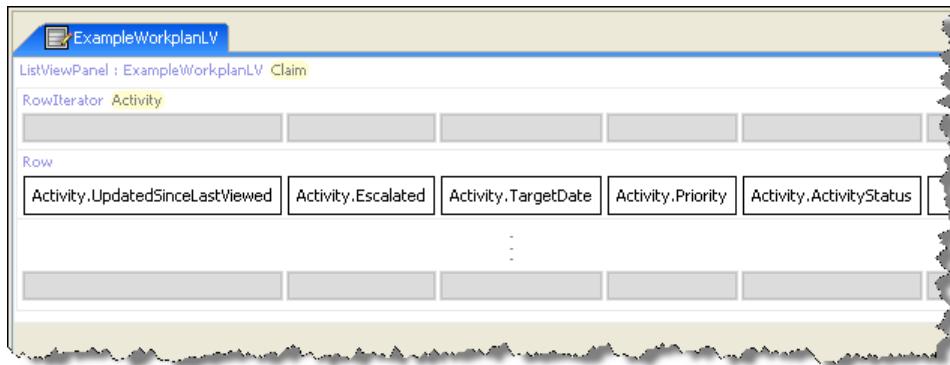
Define a row iterator with the **Row Iterator** PCF element. For example:



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 328.

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 125 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.	

Location Groups

This topic provides an introduction to location groups.

This topic includes:

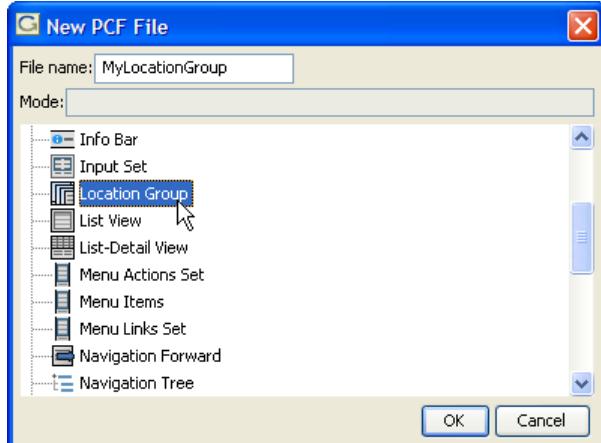
- “Location Group Overview” on page 331
- “Define a Location Group” on page 332
- “Location Groups as Navigation” on page 333

Location Group Overview

A *location group* is collection of locations. It is typically used to provide the structure and navigation for a group of related pages. ClaimCenter can automatically display the appropriate menus and other interface elements that allow users to navigate among these pages.

Define a Location Group

Define a location group with the Location Group PCF element. In the configuration → config → Page Configuration tree, click the desired folder and then right-click New → PCF file. In the New PCF File dialog, click LocationGroup, and give the location group a name. For example:



A location group must contain one or more references to another location. Any time that you navigate to the location group, ClaimCenter uses the locations defined within it to determine what page and surrounding navigation to display. The following example is the location group defined for a claim in ClaimCenter:



Location Groups as Navigation

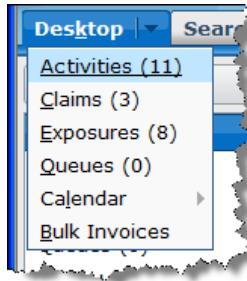
Depending on how a location group is used, ClaimCenter displays menus and other screen elements for navigating to the locations within that group. Any time that you navigate to a location group, ClaimCenter displays the first location within that group.

You can use location groups as navigation elements in the following ways:

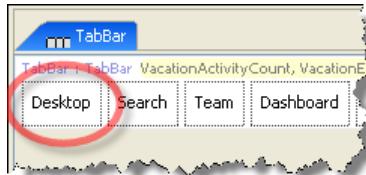
- Location Groups as Tab Menus
- Location Groups as Menu Links
- Location Groups as Screen Tabs

Location Groups as Tab Menus

A location group can be used to define the menu items that appear in a tab. As an example, consider the **Desktop** tab in ClaimCenter with the menu items as shown in the following diagram:

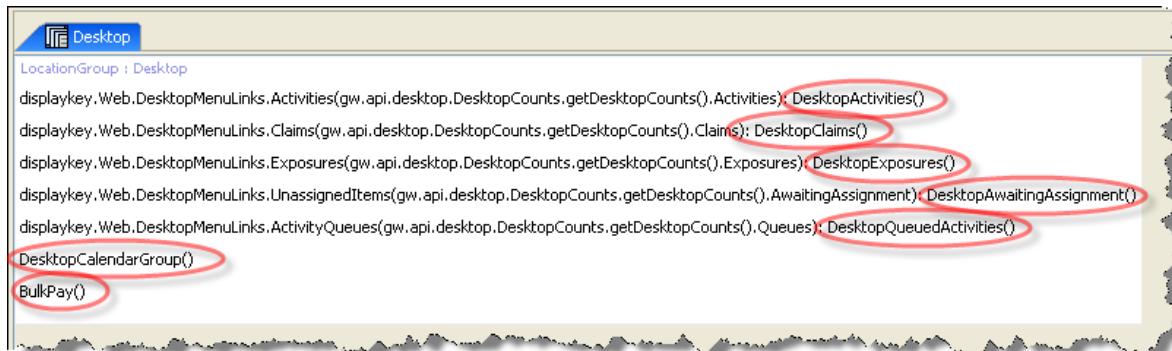


This tab is defined in the element named **TabBar** (under the **util** folder):



This tab is defined with its `action` attribute set to `Desktop.go()`. This specifies that the action to take if you click the **Desktop** tab is to go to the **Desktop** location.

This location is a location group:



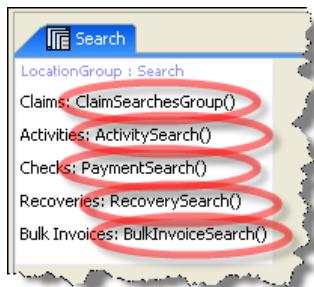
Inside this location group, there are multiple **Location Ref** elements defined, each one specifying a location. In this example, the locations referenced in the group correspond to the items in the **Desktop** menu. If the action for a tab is a location group containing more than one location, ClaimCenter adds each location in that location group to the menu in the tab.

Location Groups as Menu Links

A location group can be used to define the menu links that appear on the sidebar of the ClaimCenter interface. As an example, consider the Search page in ClaimCenter, shown in the following diagram:



The following is the definition of the Search location group:

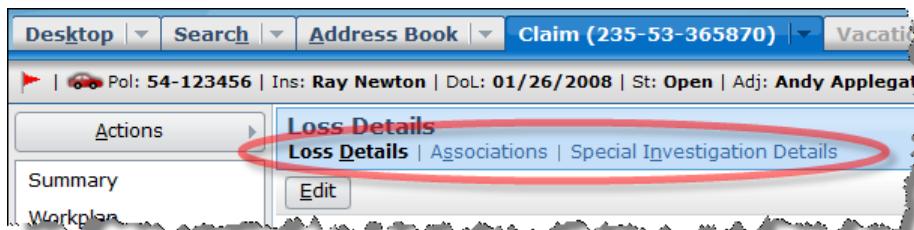


Inside this location group, there are multiple `Location_Ref` elements defined, each one specifying a location.

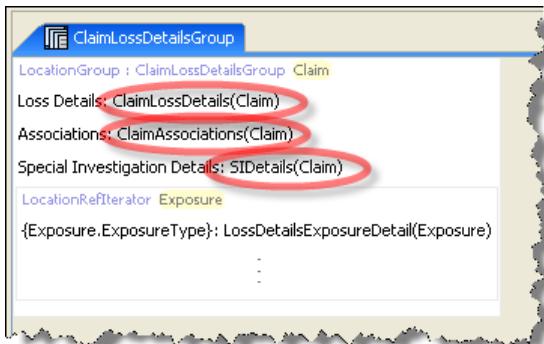
As ClaimCenter displays this location, it notices that it is a location group, and automatically creates menu links for each location within the group. Notice in this example that the `Location_Ref` elements referenced in this group correspond to the items in the menu links.

Location Groups as Screen Tabs

A location group can be used to define screen tabs that appear across the top of a screen. As an example, consider the claim Loss Details page in the following diagram:



This is a location group defined in `ClaimLossDetailsGroup` as follows:



Inside this location group, there are multiple `LocationRef` elements defined, each one specifying a location.

As ClaimCenter displays this location, it notices that it is a location group, and automatically creates screen tabs for each location within the group. Notice in this example that the `LocationRef` elements referenced in this group correspond to the items in the screen tabs.

Navigation

This topic provides an introduction to the concepts and files involved in configuring the web pages of the ClaimCenter user interface.

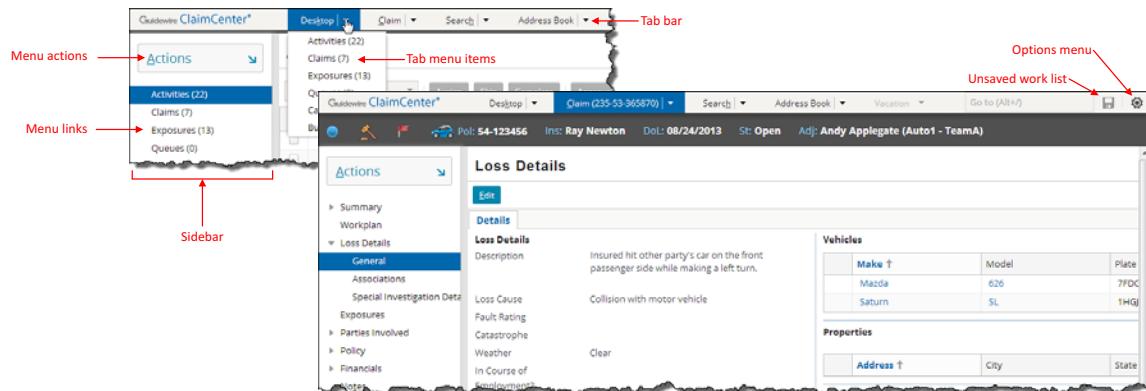
This topic includes:

- “Navigation Overview” on page 337
- “Tab Bars” on page 338
- “Tabs” on page 339

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:

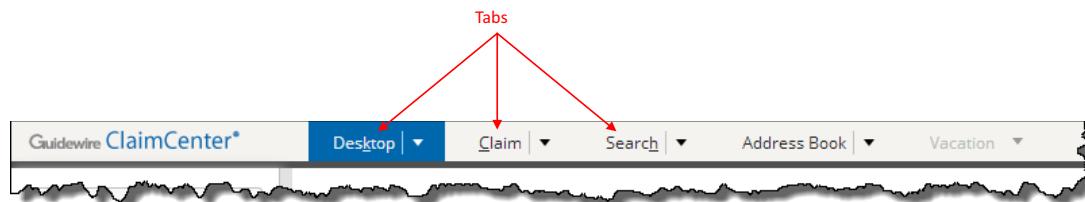


You can define the following types of navigation elements:

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 339.

Configure the Default Tab Bar

ClaimCenter 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 ClaimCenter 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, ClaimCenter 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.

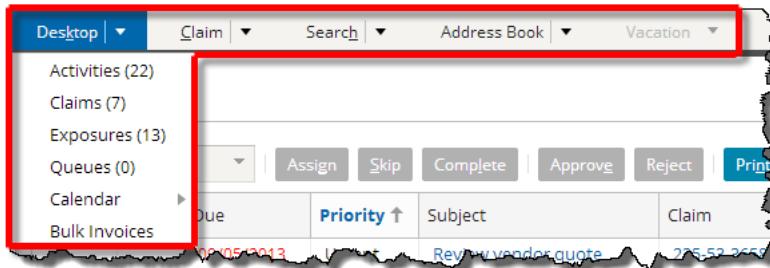
Define a Tab Bar

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

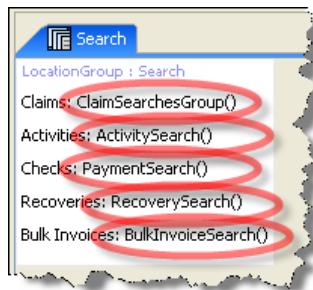
Define a Tab Menu From a Location Group

As the `action` attribute of a tab is a location group, ClaimCenter 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, ClaimCenter does not automatically add any other items to it.

Configuring Spell Check

This topic explains how to configure the spell checking feature in Guidewire ClaimCenter.

This topic includes:

- “Using the Spell Checking Feature” on page 341
- “How to Configure Spell Check” on page 342

Note: The code samples included in this topic assume the use of 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.

Using the Spell Checking Feature

Guidewire provides you with support for spell checking within the ClaimCenter application. This section explains the spell checking feature and how you can configure it.

Understanding How Spell Checking Works

Note: By default, Guidewire disables spell checking in the base configuration.

A spell check runs in response to a user action. The spell checker evaluates the text that a user enters in editable text fields or text areas and identifies any misspellings. After the spell check completes and a user either accepts or rejects any spelling suggestions, ClaimCenter saves the record as it appears. You can configure ClaimCenter to check spelling in either, or both, of the following situations:

- Anytime that a cursor leaves a text field
- Anytime that a user clicks **Check Spelling**

If you enable (and implement) the spell check functionality, ClaimCenter calls a specific check spelling method anytime that a user performs an action that initiates a spell check. Depending on how you configure your application, the functionality can become active on exit of an edited field (the field loses focus) or by clicking a **Check Spelling** button. If the user performs either action, ClaimCenter passes to the spell check the fields to check. The user then interacts with the spell checking utility to correct any spelling errors.

If you configure the **Check Spelling** button, clicking the button causes ClaimCenter to check all fields in the current page that you configure for spell checking. This check happens regardless of whether the contents of the field change or not. If you enable spell checking but do not configure this button, ClaimCenter passes only the individual fields as a user edits them. For example, suppose that you configure the **Subject** field on a note for spell checking. A user can select the field and edit it. If the user then exits the field, ClaimCenter calls the spelling checker for the **Subject** field only.

How to Configure Spell Check

Configuring ClaimCenter to support spell checking requires that you to do the following:

1. Implement a spell check utility. The utility must be available on each client machine. Your utility must provide a dictionary; ClaimCenter does not provide a dictionary.
2. Implement a `checkSpelling` method.
3. Configure the spell checker options in `config.xml`.
4. Define which application fields are subject to spell checking.

Step 1: Implement a Spell Check Utility

You can implement any tool you choose for your spell check utility, provided it meets certain functional requirements. The tool must be able to do the following:

- It must accept JavaScript calls either directly in a specialized browser window or indirectly through a control (typically, an ActiveX control).
- It must accept the parameter that ClaimCenter requires.
- It must provide an interface that a user can use to make, accept, and reject changes.

ClaimCenter does not provide a dictionary for the spell checking utility. You are responsible for providing a dictionary or otherwise ensuring the ability of your spell checking utility to determine what is—and is not—a misspelling.

Step 2: Implement a `checkSpelling` Method

If you enable spell checking through the `config.xml` file and deploy this change, ClaimCenter pulls the `SpellCheckFrame.html` as a hidden frame into ClaimCenter pages. You can find this file in the Guidewire Studio Project window under `configuration → config → web → templates → util`. File `SpellCheckFrame.html` contains JavaScript that ClaimCenter uses to call your spell checking utility. Guidewire also implements a very simple test spell checking utility in JavaScript in this file.

You can customize the `SpellCheckFrame.html` source to include or call other resources such as JavaScript or ActiveX. However, the page must always provide a JavaScript method of the following form:

```
function checkSpelling(changedFields) { ... }
```

The `changedFields` parameter is an array of HTML elements representing text fields to check. You can use the standard DOM method `document.getElementById(id)` to retrieve the actual HTML control, either a text input or a text area, and its value. Passing the ID of the control allows the spell checking utility to read and update the control value.

If you include any additional JavaScript files or ActiveX controls in the `SpellCheckFrame.html` file, you must deploy them in such a way that the `SpellCheckFrame.html` file has access to them.

Step 3: Set Spelling Checker Parameters in config.xml

You set spell check parameters in file config.xml. The following list describes the spell check parameters:

CheckSpellingOnChange	Controls whether ClaimCenter does field-by-field checking. If set to true, spell check runs immediately each time the cursor leaves a checkable text field or text area field. The default value is false.
CheckSpellingOnDemand	Controls whether the Check Spelling button appears in the ClaimCenter interface. If set to true, ClaimCenter automatically displays the button on any page that includes a spell-checked field. You do not have to edit your page configuration to display the button. By default this value is false.

You must set at least one of these fields to true to enable spell checking. If neither of these fields are set to true, spell checking does not happen regardless of whether you have implemented a spell check utility.

Step 4: Configure Text Fields for Spell Checking

You can configure text fields or text areas for spell checking. If you enable spell checking, but do not configure any fields to check, ClaimCenter does not perform spell checking. To configure a field for spell checking, set the checkSpelling attribute on the field to true.

```
<TextAreaInput  
    id="Body" value="Note.Body"  
    required="true"  
    numCols="120"  
    numRows="10"  
    checkSpelling="true"  
    editable="Note.isBodyEditable()"/>
```

Some of the commonly configured text fields for spell checking include long fields (notes and descriptions) and text fields relating to litigation. ClaimCenter validates your PCF files to verify that the fields on which you set the checkSpelling attribute are text fields.

Configuring Search Functionality

This topic describes how to configure the Guidewire ClaimCenter search functionality. ClaimCenter provides a **Search** tab that you can use to search for specific entities. You can customize the **Search** tab to add new search criteria or modify or remove existing criteria. This topic also describes how the search feature integrates with the underlying data model and the XML and PCF configuration files you need to modify to customize the **Search** tab.

You can only customize the search associated with the **Search** tab (and documents). You cannot customize specialized searches for users and groups, for example.

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.

This topic includes:

- “Search Overview” on page 345
- “Database Search Configuration” on page 347
- “Free-text Search Configuration” on page 360

Search Overview

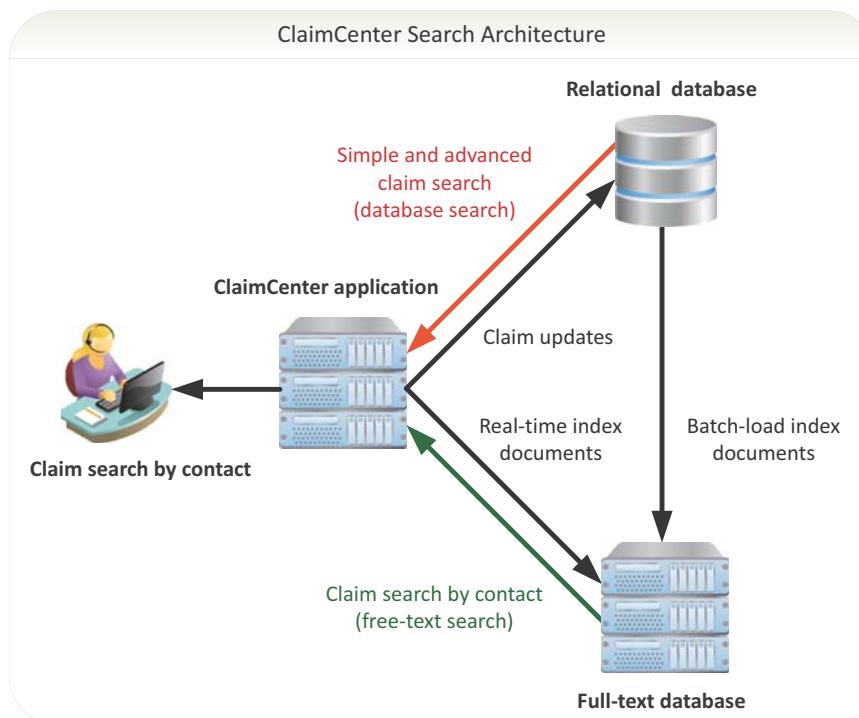
ClaimCenter provides two types of search:

- **Database search** – Searches the relational database for claims, contacts, activities, checks, recoveries, and bulk invoices by using the Structured Query Language (SQL). You access these searches from the **Search** tab. ClaimCenter also includes database search from screens besides those accessed through the **Search** tab. For example, you can do a database search for policies from the New Claim screen.
- **Free-text search** – Searches an external full-text database for claims by claim contact using the APIs of an external full-text search engine. You access free-text search from the **Search** → **Claims** → **Search by Contact** screen.

Database search is fully enabled by default. Users can choose to include archived claims with each database search request.

Free-text search is available as an option that you must enable and configure. Users can search only for claims by claim contact with free-text search. Free-text search results never include archived claims.

IMPORTANT Guidewire does not support configuring ClaimCenter for free-text search of entity types other than claim contacts.



If you enable free-text search, the **Search → Claims** menu displays three options:

- **Simple Search, Advanced Search**—Displays fields on which to search for claims using database search.
- **Search by Contact** – Displays fields on which to search for claims using claim contacts and free-text search.

If you disable free-text search, the **Search → Claims** menu displays only the **Simple** and **Advanced** options.

Reasons users choose include:

- Users want to search for claims by claim contact with commonly used criteria and receive results quickly, which the **Search by Contact** screen provides.
- Users want to search claims with highly targeted criteria, which requires the **Simple Search** screen.
- Users want to search claims with partial names, phonetic names, or sounds-like names, which the **Search by Contact** screen provides.
- Users want to search archived claims, which requires the **Advanced Search** screen.

See also

- “Database Search Configuration” on page 347
- “Free-text Search Configuration” on page 360

Database Search Configuration

This section describes how to configure database search. Topics include:

- “ClaimCenter Database Search Functionality” on page 347
- “Configuring ClaimCenter Database Search” on page 348
- “Deploying Customized Database Search Files” on page 355
- “Steps in Customizing a Database Search” on page 355

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.

ClaimCenter Database Search Functionality

To search for a specific entity, select the **Search** tab from the ClaimCenter interface. In the base configuration, you can search for the following:

- Claims
- Activities
- Checks
- Recoveries
- Bulk Invoices

During a search, ClaimCenter uses only those fields on the form for which you enter data. For example, if you search for a **Claimant** and enter a **Last Name** but not a **Policy Number**, ClaimCenter omits **Policy Number** from the search.

For each search, ClaimCenter requires specific fields while others are optional. You can add, modify or remove optional fields. You cannot add required fields. Also, do not modify or remove required fields specified in the ClaimCenter base search configuration.

The PCF files that define a particular search page reflect this division. (You can find the search pages in Studio under **PCF → search**.) Each search detail view references two subviews, each contained within their own PCF. For example, **ActivitySearchDV** defines the detail view and includes the following two subviews:

- **ActivitySearchRequiredInputSet**
- **ActivitySearchOptionalInputSet**

During a search, the Guidewire application uses a search criteria object. Every field on the **Search** page maps back to an attribute on the relevant search criteria entity. For example, in the **Activity** search screen, the value that you set in the **Assigned To User** field maps to **ActivitySearchCriteria.AssignedToUser** in the **ActivitySearchRequiredInputSet** PCF file.

Note: Search criteria entities are *virtual* entities. Virtual entities have no underlying table in the ClaimCenter database. Rather, they are non-persistent and only exist within the session in which you use them.

In most cases, each attribute on the search criteria entity maps to one attribute on the key entity. In the case of drop-down widgets, however, the search criteria object contains an attribute that points to an array. The array can point to multiple attributes on the key entity. The **search-config.xml** file (under **Other Resources**) maps search criteria to the target entities.

The fields in a search form correspond to entity attributes in the data model. You can customize the search process to search by an attribute on the key entity or any of its related objects. For example, to use the **Activity** search screen again, the value that you set in the **Assigned To User** field maps to

`ActivitySearchCriteria.AssignedToUser`. This, in turn, maps (through `search-config.xml`) to `Activity.AssignedUser`.

Configuring ClaimCenter Database Search

You use the `search-config.xml` file to define a mapping between the key data entities and the search criteria entities. The entries in the file have the following basic structure:

```
<CriteriaDef entity="name" targetEntity="name">
    <Criterion property="attributename" targetProperty="attributename" matchType="type"/>
    <CriterionChoice property="name" init="value_or_expression">
        <Option label="name" targetProperty="attributename"/>
        <Option label="name" targetProperty="attributename"/>
        ...
    </CriterionChoice/>
    <ArrayCriterion property="attributename" targetProperty="attributename"
        arrayMemberProperty="attributename"/>
</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 CriterionChoice ArrayCriterion	Specifies the mapping from a search criteria entity to the target entity on which to search. WARNING Do not add new CriteriaDef elements to <code>search-config.xml</code> . Instead, modify only the contents of existing CriteriaDef elements. The ClaimCenter search code uses these criteria definitions as it builds a query. If you add a new CriteriaDef element, you can cause the search engine to work incorrectly.
Criterion		Specifies how ClaimCenter 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.
CriterionChoice	Option	Defines specialized properties in the search criteria that can match against a number of target fields.
Option		Describes a single choice within a criterion choice.
ArrayCriterion		Specifies that the search query uses a simple join against an array entity.

See also

- “The `<CriteriaDef>` Element” on page 349.
- “The `<Criterion>` Subelement” on page 350.
- “The `<CriterionChoice>` Subelement” on page 351.
- “The `<ArrayCriterion>` Subelement” on page 354.
- For information on configuring claim search by claim contacts, see “Free-text Search Configuration” on page 360.
- For information on configuring search for contacts, see “Searching for Contacts” on page 83 in the *Contact Management Guide*.

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:

<CriteriaDef> 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:

- Claim
- ClaimInfo
- ClaimRpt

The ClaimRpt table contains denormalized claim financials information. By mapping to this table, ClaimCenter increases search performance for financial related claim searches. An example of this type of search is searching for a claim with a specific open reserve amount. By mapping to ClaimRpt, ClaimCenter avoids calculating financial values within the search query itself.

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 ClaimCenter 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:

<CriteriaDef> subelement	Description
Criterion	Performs simple, one-to-one mapping between a criteria entity attribute and a target entity attribute.
CriterionChoice	Matches an attribute on the search criteria entity against any one of a number of target attributes.
ArrayCriterion	Creates a simple join against an array entity.

The following table lists each search criteria object specified in `search-config.xml` and its corresponding entity objects in ClaimCenter:

Search criteria object	Entity
ActivitySearchCriteria	Activity
Address	Address
BulkInvoiceSearchCriteria	BulkInvoice
CCNameCriteria	Contact
ClaimInfoCriteria	ClaimInfo
ClaimSearchCriteria	Claim ClaimInfo ClaimRpt
DocumentSearchCriteria	Document

Search criteria object	Entity
PaymentSearchCriteria	Check CheckRpt
RecoverySearchCriteria	Recovery
UserSearchCriteria	Attribute AttributeUser AuthorityLimitProfile User

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:

<Criterion> attribute	Required	Description
property	Yes	The name attribute on the criteria entity. The search engine uses this value to fetch user's input value from the criteria entity.
matchType	Yes	This attribute is dependant on the data type of the targetProperty. See the following table for possible values.
forceEqMatchType	No	<p>The name of a Boolean property on the criteria entity:</p> <ul style="list-style-type: none"> • If this attribute evaluates to true, then the Criterion uses an eq (equality) match. • If this attribute evaluates to false, then the Criterion uses the matchType 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 startsWith match for StringProperty unless the FlagProperty on the criteria entity is true, in which case, the match uses an eq match type.</p>
targetProperty	No	<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
contains		String	IMPORTANT Guidewire strongly recommends that you avoid using the contains match type, if at all possible. The contains match type is the most expensive type in terms of performance.
eq	equals	Numeric or Date	
ge	greater than or equal	Numeric or Date	
gt	greater than	Numeric or Date	
le	less than or equal	Numeric or Date	

Match type	Evaluates to	Use with data type	Comments
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 ClaimCenter search screens.

The <CriterionChoice> Subelement

It is possible for a search to be more complex than a simple one-to-one mapping. For example, the ClaimCenter Search Claims page contains a **Search For Date** field. Properties such as **Search for Date** complicate search configuration because a single column in the search criteria can match against one of several columns in the target. To handle these cases, you use the <CriterionChoice> element, a subelement of the <CriteriaDef> element. A <CriterionChoice> parameter matches an attribute on the search criteria entity against any one of a number of target attributes.

A <CriterionChoice> element uses the following syntax:

```
<CriterionChoice property="attributeName" init="valueOrExpression">
  ...
</CriterionChoice>
```

These attributes have the following definitions:

<CriterionChoice> attribute	Required	Description
property	Yes	One of the following: <ul style="list-style-type: none"> • DateCriterionChoice – Use to specify the options available to restrict a date search. • ArchiveDateCriterionChoice – Use to specify the options available to restrict a date search on an archived object. • FinancialCriterion – Use to specify the options available to restrict a financial field search.
init	No	Gosu string that DateCriterionChoice and ArchiveDateCriterionChoice use to initialize the criterion choice.

A <CriterionChoice> element can have the following subelement:

<CriterionChoice> subelement	Description
Option	Describes a single choice within a criterion choice.

You can specify a single Option subelement or many. If you specify a single Option, there are no choices for specifying the criteria. ClaimCenter limits the choice to the single option that you specify.

The <Option> subelement contains the following attributes:

<Option> attribute	Required	Description
label	Yes	Display key that indicates the choice. ClaimCenter uses this text to display the option to the user in a select control.
targetProperty	No	Target column (field) against which ClaimCenter matches the user-input value. Do not use a virtual property as search field on an entity.
WARNING The targetProperty attribute is optional, but Guidewire requires you to specify this attribute if you add a new <Option> element. Omitting this attribute on a new option can cause ClaimCenter to not operate properly.		

Setting the Property Attribute to DateCriterionChoice

You use the <DateCriterionChoice> element as part of a larger search criteria <CriteriaDef> element. The <DateCriterionChoice> element encapsulates the information entered by the user to restrict the search to a particular date range. The following example from the ClaimCenter ClaimSearchCriteria <CriteriaDef> element configured on the Claim entity illustrates the use of this attribute:

```

<CriteriaDef entity="ClaimSearchCriteria" targetEntity="Claim">
  ...
  <CriterionChoice property="DateCriterionChoice"
    init="DateCriterionChoice.SearchType = "claim";
    DateCriterionChoice.DateSearchType = "fromlist";
    DateCriterionChoice.DateRangeChoice = "n365";
    DateCriterionChoice.ChosenOption = "Java.Criterion.Option.Claim.LossDate"*>
    <Option label="Java.Criterion.Option.Claim.LossDate" targetProperty="LossDate"/>
    <Option label="Java.Criterion.Option.Claim.ReportedDate" targetProperty="ReportedDate"/>
    <Option label="Java.Criterion.Option.Claim.CloseDate" targetProperty="CloseDate"/>
    <Option label="Java.Criterion.Option.Claim.CreateDate" targetProperty="CreateTime"/>
  </CriterionChoice>
  ...
</CriteriaDef>
```

The `init` attribute specifies how to restrict the date field. The user can restrict the date range either through a typelist of preset ranges (such as next 30 days) or through a specific start and end date. The `init` attribute sets the following values:

Value	Required	Description
SearchType	Yes	A value from the <code>SearchObjectType</code> typelist. This value determines the entity on which to search.
DateSearchType	Yes	A value from the <code>DateSearchType</code> typelist. In the base configuration, the possible values are: <ul style="list-style-type: none"> • <code>enteredrange</code> • <code>fromlist</code> ClaimCenter renders a widget for <code>DateCriterionChoice</code> that you can use to enter a date range either from a predefined list (<code>fromlist</code>) or by manually entering a range (<code>enteredrange</code>). This value determines which method is the default choice in the widget.
DateRangeChoice	Yes	A value from the <code>DateRangeChoiceType</code> typelist. This value determines the default date range. ClaimCenter filters the available ranges by the <code>DateSearchType</code> field. You use this only if <code>DateSearchType</code> is set to <code>fromlist</code> .
ChosenOption	No	The default date on which to search: <ul style="list-style-type: none"> • If you use this field, set it to one of the option labels contained in this <code>CriterionChoice</code> element. • If you do not set this option, then this value defaults to <code><none selected></code>.

This `<CriterionChoice>` definition sets the available `<Option>` elements on which to search. In this case:

- Loss date
- Reported date
- Close date
- Create time

The following limitations apply:

1. You cannot specify a match type for criterion choice entities. As their matching algorithm is built into the entity, you cannot configure it.
2. Guidewire initializes `<DateCriterionChoice>` properties for the major searches in the base configuration `search-config.xml` file. This configuration limits searches by date to improve performance on large databases, in which searching a very large number of claims or activities (or both) can be resource intensive. Typically, most users do not expect their searches to be date limited. However, these limitations are necessary for acceptable performance.

Setting the Property Attribute to FinancialCriterion

You use the `<FinancialCriterion>` element as part of a larger search criteria `<CriteriaDef>` element. The `<FinancialCriterion>` element encapsulates the information entered by the user to restrict the search to entities with a money field within a given range.

For example, in ClaimCenter, the claim search page uses a `<FinancialCriterion>` element with multiple options:

```
<CriteriaDef entity="ClaimSearchCriteria" targetEntity="ClaimRpt">
  <CriterionChoice property="FinancialCriterion">
    <Option label="Java.Criterion.Option.Claim.OpenReserves" targetProperty="OpenReserves"/>
    <Option label="Java.Criterion.Option.Claim.RemainingReserves" targetProperty="RemainingReserves"/>
    <Option label="Java.Criterion.Option.Claim.TotalPayments" targetProperty="TotalPayments"/>
    <Option label="Java.Criterion.Option.Claim.FuturePayments" targetProperty="FuturePayments"/>
    <Option label="Java.Criterion.Option.Claim.TotalIncurredGross"/>
    <Option label="Java.Criterion.Option.Claim.TotalIncurredNet"/>
  </CriterionChoice>
</CriteriaDef>
```

The ClaimCenter payment search page uses a single option:

```
<CriterionChoice property="FinancialCriterion">
  <Option label="Java.Criterion.Option.Payment.GrossAmount" targetProperty="GrossAmount"/>
</CriterionChoice>
```

The `<ArrayCriterion>` Subelement

An `<ArrayCriterion>` element instructs ClaimCenter to add a simple join against an array entity to the search query. You add the `<ArrayCriterion>` subelement to a `<CriteriaDef>` element. An `<ArrayCriterion>` subelement uses the following syntax:

```
<CriteriaDef>
  <ArrayCriterion property="someName" targetProperty="someTargetProperty"
    arrayMemberProperty="someArrayMember"/>
  ...
</CriteriaDef>
```

These attributes have the following definitions:

<code><ArrayCriterion></code> attribute	Required	Description
<code>property</code>	Yes	The name attribute given for the column (field) on the search criteria entity. The search engine uses this value to fetch the user-entered input value on the criteria entity.
<code>targetProperty</code>	Yes	The name attribute of the array on the target entity.
<code>arrayMemberProperty</code>	Yes	The name attribute of a column (field) in the array member type. For example, if <code>targetProperty</code> refers to an array of X-type entities, then <code>arrayMemberProperty</code> is a column on an instance of Entity X.

For example, in ClaimCenter, suppose that you add a custom array called `ClaimCodes` to `Claim` and that each member of the `ClaimCodes` array is a `ClaimCode` object. Further, `ClaimCode` contains just two fields, the required back reference to the `Claim` table and a `Code` field of type `varchar`. For this example, the `<ArrayCriterion>` element to add to `search-config.xml` looks similar to the following:

```
<ArrayCriterion property="ClaimCode" targetProperty="ClaimCodes" arrayMemberProperty="Code"/>
```

For complete details of this example, see “Adding an Optional Array Search Field” on page 357.

The following limitations apply:

1. The array member column must not allow duplicate values. In the given example, no single `Claim` can have two `ClaimCode` entries with the same code. Violating this restriction causes the same claim to appear multiple times in the search results.
2. For performance reasons, it is important to have two unique indexes on the array table. These indexes enforce the first restriction, no duplicate values, and also help to make the search perform acceptably.
 - The first index must contain the back reference to the owner of the array—the `Claim` ID in the example—and the array member column itself—`Code` in the example.
 - The second index must contain the same two columns, but in reverse order.

For example:

```
<foreignkey name="ClaimID" fkentity="Claim" nullok="false" exportable="false" desc="Related claim."/>
<index name="ind1" unique="true">
  <indexcol name="ClaimID" keyposition="1"/>
  <indexcol name="Code" keyposition="2"/>
</index>
<index name="ind2" unique="true">
  <indexcol name="Code" keyposition="1"/>
  <indexcol name="ClaimID" keyposition="2"/>
</index>
```

3. It is not possible to specify a `matchType` property on this criterion. Guidewire supports equality matching only for this type of criterion.

Deploying Customized Database Search Files

Guidewire recommends that you first make your search customization file changes in your development environment. You must then create a .war or .ear file and move your changes to the production server.

The ClaimCenter production server validates your search configuration every time that the server starts. If the validation fails, the server does not start. The production server validates the following:

- The `CriteriaDef` entity and `targetEntity` attributes reference real entities.
- The `targetEntity` type is a persistent entity.
- The `targetProperty` attributes reference searchable properties on the target entity, except for those on `ArrayCriterion` elements that must reference an array column.
- The type of each `property` attribute on a `Criterion` element matches the type of its corresponding `targetProperty`. (For example, they are both strings or both numbers.)
- All `Criterion` match types are suitable for the criterion property. (For example, you can only use `startsWith` for string properties.)
- All `CriterionChoice` property attributes specify foreign key links to entities that implement the `SearchCriterionChoice` interface.
- All `CriterionChoice init` property attributes execute without errors against a newly created criterion choice entity of the appropriate type.
- All `Option` label attributes reference valid display keys.
- All `ArrayCriterion arrayMemberProperty` attributes reference searchable properties on the array member entity.

Steps in Customizing a Database Search

You can customize search in several different ways. For example, you can modify the existing optional search criteria or you can add your own new, optional search criteria. However, you cannot add new, required search criteria or modify existing required search criteria.

This topic includes:

- “Working with Optional Search Criteria” on page 355
- “Adding an Optional Search Field” on page 356
- “Adding an Optional Array Search Field” on page 357

See also

- “ClaimCenter Database Search Functionality” on page 347
- “Configuring ClaimCenter Database Search” on page 348
- “The `<CriteriaDef>` Element” on page 349

Working with Optional Search Criteria

To add an entirely new optional search criterion, you must do the following:

- Extend an existing key entity or one of its related entities. This is optional if you only wish to use an existing base column as a searchable column.
- Add an extension field on the search criteria entity.
- Add the new search `Criterion`, `CriterionChoice` or `ArrayCriterionChoice` element to file `search-config.xml`.

- Configure the new widget in the search PCF file.

Note: Do not attempt to change the match type of a non-required criterion that exists in the base configuration. Although not strictly prohibited, changing the match type of a non-required criterion has no effect because ClaimCenter ignores the change.

Adding an Optional Search Field

In this example, you add an extension field on the `Claim` entity called `PercentComplete`. You can use this field to search on a claim that is 90% complete.

To add an optional search field

- Create file `Claim.etx` in `configuration → config → Extensions → Entity`, if one does not exist already. Enter the following information in the `Claim` extension file.

```
<extension entityName="Claim">
  <column name="PercentComplete" type="percentagedec" desc="Percentage complete on the Claim"/>
</extension>
```

This action extends the `Claim` entity and adds a `PercentComplete` field on the `Claim` object. See “Extending a Base Configuration Entity” on page 213 for details of this process, if necessary.

- Create file `ClaimSearchCriteria.etx` in `configuration → config → Extensions → Entity`, if one does not exist already. Enter the following information:

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel"
  entityName="ClaimSearchCriteria">
  <column desc="Percentage of claim that is complete"
    name="PercentComplete"
    type="percentagedec"/>
</extension>
```

This action adds the new claim search criteria field to the `ClaimSearchCriteria` entity.

- Open file `search-config.xml` for editing and add a `<Criterion>` element to the `<CriteriaDef>` element for the `ClaimSearchCriteria` entity configured against the `Claim` entity:

```
<CriteriaDef entity="ClaimSearchCriteria" targetEntity="Claim">
  <Criterion property="ClaimNumber" matchType="eq"/>
  <Criterion property="AssignedToUser" targetProperty="AssignedUser" matchType="eq"/>
  ...
  <!-- Example extension -->
  <Criterion property="PercentComplete" matchType="ge"/>
</CriteriaDef>
```

Note: Guidewire recommends that you ensure that appropriate indexes are in place if you change the search criteria. For example, if `PercentComplete` is the most restrictive equality condition in your search implementation, then consider adding an index with this column as the leading key column.

- Within file `search-config.xml`, increment the file version number before you save the file. Although there is no strict requirement that you do so, Guidewire recommends that you increment the version number if you modify this file.

```
<?xml version="1.0" encoding="UTF-8"?>
<SearchConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="search-config.xsd"
  version="1">
  ...

```

- Add a display key for the Percent Complete field label:

- Navigate to `configuration → config → Localizations → Lang` in Studio, and open the `display.properties` file.
- Find the display key entries that begin with `JSP.ClaimSearch.Claims`, and add the following line:

```
JSP.ClaimSearch.Claims.PercentComplete = Percent Complete
```

6. Edit file `ClaimSearchOptionalInputSet.pcf` and add a new **Percent Complete** field. You must identify this new field as editable. In brief, you need to do the following:

- a. Navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **search** → **claims** and open `ClaimSearchOptionalInputSet.pcf` for editing.
- b. Drag an **Input** widget on to the page. You can put it in the **Optional parameters** section, directly under the **Claim Status** field, for example.
- c. Set the following values in the **Properties** tab at the bottom of the screen:

<code>editable</code>	<code>true</code>
<code>id</code>	<code>Completion</code>
<code>label</code>	<code>displaykey.JSP.ClaimSearch.Claims.PercentComplete</code>
<code>required</code>	<code>false</code>
<code>value</code>	<code>ClaimSearchCriteria.PercentComplete</code>

See “Using the PCF Editor” on page 295 for details of how to modify a PCF page if necessary.

7. Save all your work.

8. Restart Guidewire Studio. Although there is no strict requirement that you restart Studio to complete the configuration, it is a good practice. For example, restarting Studio often catches simple typing errors.

9. Regenerate the application SOAP and Java APIs and the *Data Dictionary*. Although there is no strict requirement that you regenerate these items, Guidewire recommends that you do so as a good practice. Regenerating the APIs propagates your configuration changes to the APIs in the event that they are necessary.

Do the following:

- a. Open a command window and navigate to the application `bin` directory.
- b. Execute the following commands:

```
gwcc regen-java-api  
gwcc regen-soap-api  
gwcc regen-dictionary
```

For more information on these commands, see “Build Tools” on page 106 in the *Installation Guide*.

Note: The `regen-soap-api` command regenerates RPC-Encoded SOAP APIs only. To regenerate WS-I APIs, stop and restart the application. ClaimCenter regenerates the APIs automatically.

10. Restart the application server.

11. Log into Guidewire ClaimCenter and navigate to the **Search Claims** → **Advanced Search** page. Verify that your optional claim search field exists.

The claim search page contains your new field as a searchable option. Of course, to support this new functionality, you need to provide a way to assign a percentage complete on a claim. For example, in ClaimCenter, you might provide a new field on one of the claim screens.

Adding an Optional Array Search Field

Suppose you want the ability to enter a claim code from a list of possible codes and have the search return any claim whose claim code matches. To support this behavior, you add a new, searchable array of codes to the optional search fields. Thus, you have the following:

- `ClaimCodes` – A custom array of `ClaimCode` objects attached to `Claim`.
- `ClaimCode` – A member of the `ClaimCodes` array. Each member contains two fields, the required foreign key reference to the `Claim` table and a `Code` field of type `varchar`.

Entering a value for the `ClaimCode` field on the search screen returns all `Claim` objects whose `ClaimCodes` array contains an entry with a matching `Code` value. To support this functionality, you need to provide a way to assign a code on a claim. For example, in ClaimCenter, you can provide a new field on one of the claim screens.

Note: The code samples included in this topic assume the use of 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.

To add an optional array search field

1. Create file `ClaimCode.eti` in **configuration** → **config** → **Extensions** → **Entity**. Enter the following information:

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
    desc="Code associated with a claim."
    entity="ClaimCode"
    exportable="true"
    extendable="true"
    table="claimcode"
    type="retireable">
    <implementsEntity name="Extractable"/>
    <column name="Code" type="varchar" desc="code ">
        <columnParam name="size" value="30"/>
    </column>
    <foreignkey name="ClaimID" fkentity="Claim" nullok="false" exportable="false" desc="Related claim."/>
    <index name="ind1" unique="true">
        <indexcol name="ClaimID" keyposition="1"/>
        <indexcol name="Code" keyposition="2"/>
    </index>
    <index name="ind2" unique="true">
        <indexcol name="Code" keyposition="1"/>
        <indexcol name="ClaimID" keyposition="2"/>
    </index>
</entity>
```

This action creates a new `ClaimCode` entity to use as a member in an array of `ClaimCode` objects. Each `ClaimCode` entity contains two fields, the (required) foreign key reference to the `Claim` table and a `Code` field of type `varchar`.

2. Create file `Claim.etx` in **configuration** → **config** → **Extensions** → **Entity**, if one does not exist already. Enter the following information in the `Claim` extension file.

```
<array arrayentity="ClaimCode"
    desc="Set of claim codes associated with this claim."
    exportable="true"
    name="ClaimCodes"
    owner="true"/>
```

This action extends the `Claim` entity and adds a `ClaimCodes` array to the `Claim` object. See “Extending a Base Configuration Entity” on page 213 for details of this process, if necessary.

3. Create file `ClaimSearchCriteria.etx` in **configuration** → **config** → **Extensions** → **Entity**, if one does not exist already. Enter the following information:

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel"
    entityName="ClaimSearchCriteria">
    <column name="ClaimCode" type="varchar" desc="Claim Code">
        <columnParam name="size" value="30"/>
    </column>
</extension>
```

This action adds the new claim search criteria field to the `ClaimSearchCriteria` entity.

4. Open file `search-config.xml` for editing and add the following to the `<CriteriaDef>` element for the `ClaimSearchCriteria` entity configured against the `Claim` entity:

```
<ArrayCriterion property="ClaimCode" targetProperty="ClaimCodes" arrayMemberProperty="Code"/>
```

5. Within file `search-config.xml`, increment the file version number before you save the file. Although there is no strict requirement that you do so, Guidewire recommends that you increment the version number if you modify this file.

```
<?xml version="1.0" encoding="UTF-8"?>
<SearchConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:noNamespaceSchemaLocation="search-config.xsd"
version="1">
```

6. Add a display key for the **Claim Code** 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 **JSP.ClaimSearch.Claims**, and add the following line:
JSP.ClaimSearch.Claims.ClaimCode = Claim Code

7. Edit file **ClaimSearchOptionalInputSet.pcf** and add a new **Claim Codes** field. You must identify this new field as editable. In brief, you need to do the following:

- a.** Navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **search** → **claims** and open **ClaimSearchOptionalInputSet.pcf** for editing.
- b.** Drag an **Input** widget on to the page. You can put it in the **Optional parameters** section, directly under the **Claim Status** field, for example.
- c.** Set the following values in the **Properties** tab at the bottom of the screen:

editable	true
id	ClaimCode
label	displaykey.JSP.ClaimSearch.Claims.ClaimCode
required	false
value	ClaimSearchCriteria.ClaimCode

See “Using the PCF Editor” on page 295 for details of how to modify a PCF page if necessary.

8. Save all your work.

9. Restart Guidewire Studio. Although there is no strict requirement that you restart Studio to complete the configuration, it is a good practice. For example, restarting Studio often catches simple typing errors.

10. Regenerate the application SOAP and Java APIs and the *Data Dictionary*. Although there is no strict requirement that you regenerate these items, Guidewire recommends that you do so as a good practice. Regenerating the APIs propagates your configuration changes to the APIs in the event that they are necessary.

Do the following:

- a.** Open a command window and navigate to the application **bin** directory.
- b.** Execute the following commands:

```
gwcc regen-java-api
gwcc regen-soap-api
gwcc regen-dictionary
```

For more information on these commands, see “Build Tools” on page 106 in the *Installation Guide*.

Note: The **regen-soap-api** command regenerates RPC-Encoded SOAP APIs only. To regenerate WS-I APIs, stop and restart the application. ClaimCenter regenerates the APIs automatically.

11. Restart the application server.

12. Log into ClaimCenter and navigate to the **Search Claims** → **Advanced Search** page. Verify that your optional search field exists.

The search page now contains a field on which you can search by claim codes. Of course, to support this functionality, you need to provide a way to assign a code on a claim. For example, in ClaimCenter, you can provide a new field on one of the claim screens.

Free-text Search Configuration

This topic describes how to enable and configure free-text search for ClaimCenter. It includes:

- “Overview of Free-text Search” on page 360
- “Free-text Search System Architecture” on page 360
- “Enabling Free-text Search in ClaimCenter” on page 363
- “Configuring the Guidewire Solr Extension for Integration with ClaimCenter” on page 364
- “Configuring the Free-text Batch Load Command” on page 368
- “Configuring Free-text Search for Indexing and Searching” on page 367
- “Configuring the Search by Contact Screen for Free-text Search” on page 368
- “Modifying Free-text Search for Additional Fields” on page 369

See also

- “Free-text Search Setup” on page 85 in the *Installation Guide*
- “Free-text Search Integration” on page 595 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 ClaimCenter.
- **Embedded** – Supported only in development environments, the Guidewire Solr Extension runs automatically as part of ClaimCenter in the application server instance that runs ClaimCenter. With embedded operation, the Guidewire Solr Extension does not run as a separate application.

Free-text Search System Architecture

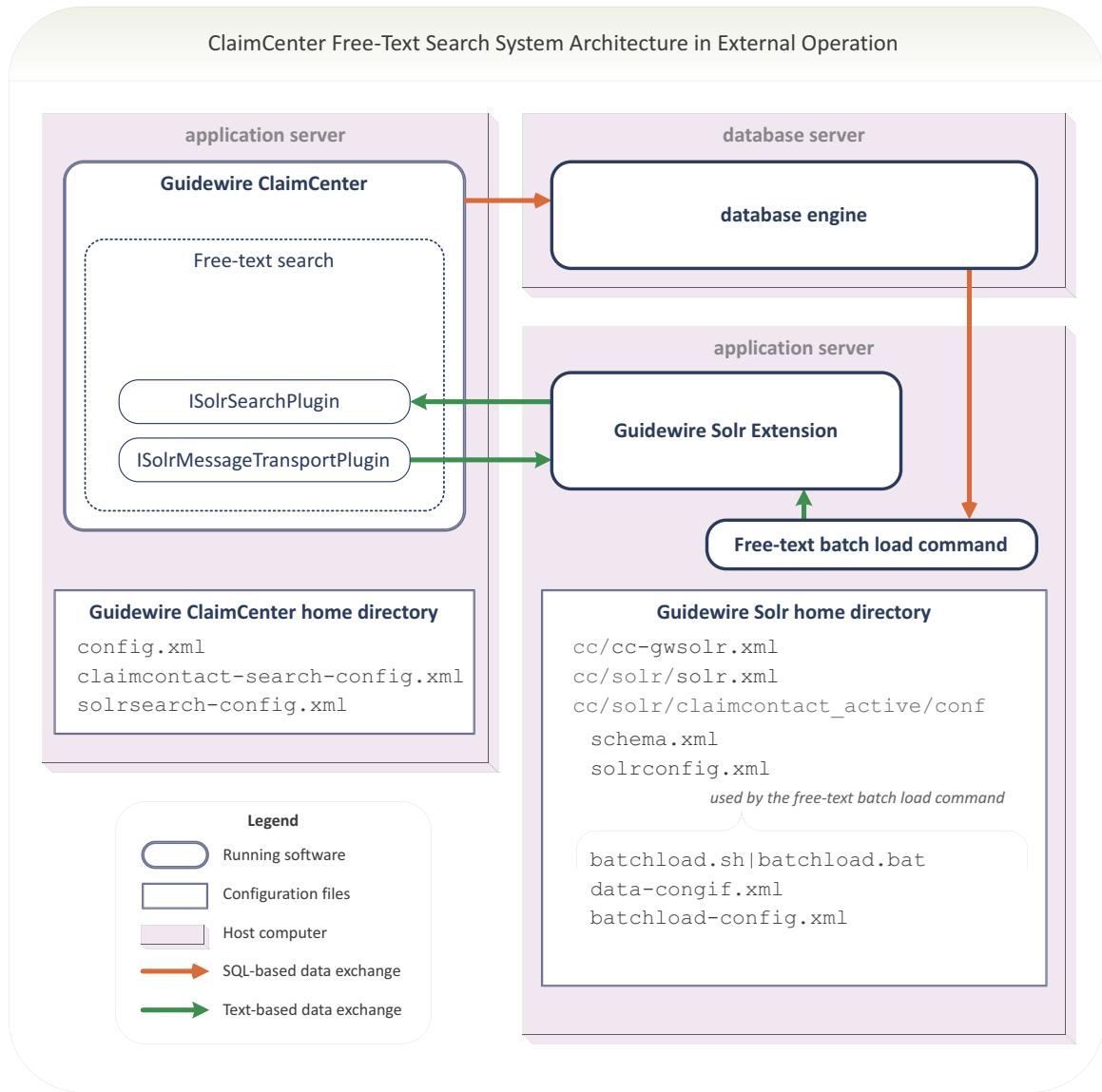
The system architecture of the Guidewire free-text search feature comprises the following components:

- The Guidewire ClaimCenter application
- The Guidewire Solr Extension, a modified version of the Apache Solr full-text search engine
- The Guidewire free-text batch load command
- The Solr Data Import batch process that you run from the **Batch Process** page on the **Internal Tools** tab in the ClaimCenter application. The Solr Data Import batch process is an alternative to running the free-text batch load command. You must use the Solr Data Import batch process instead of the batch load command whenever you configure free-text search for embedded operation.

The components of the free-text search feature depend on configuration parameters and configuration files in two primary locations: the ClaimCenter home directory and a separate Guidewire Solr home directory.

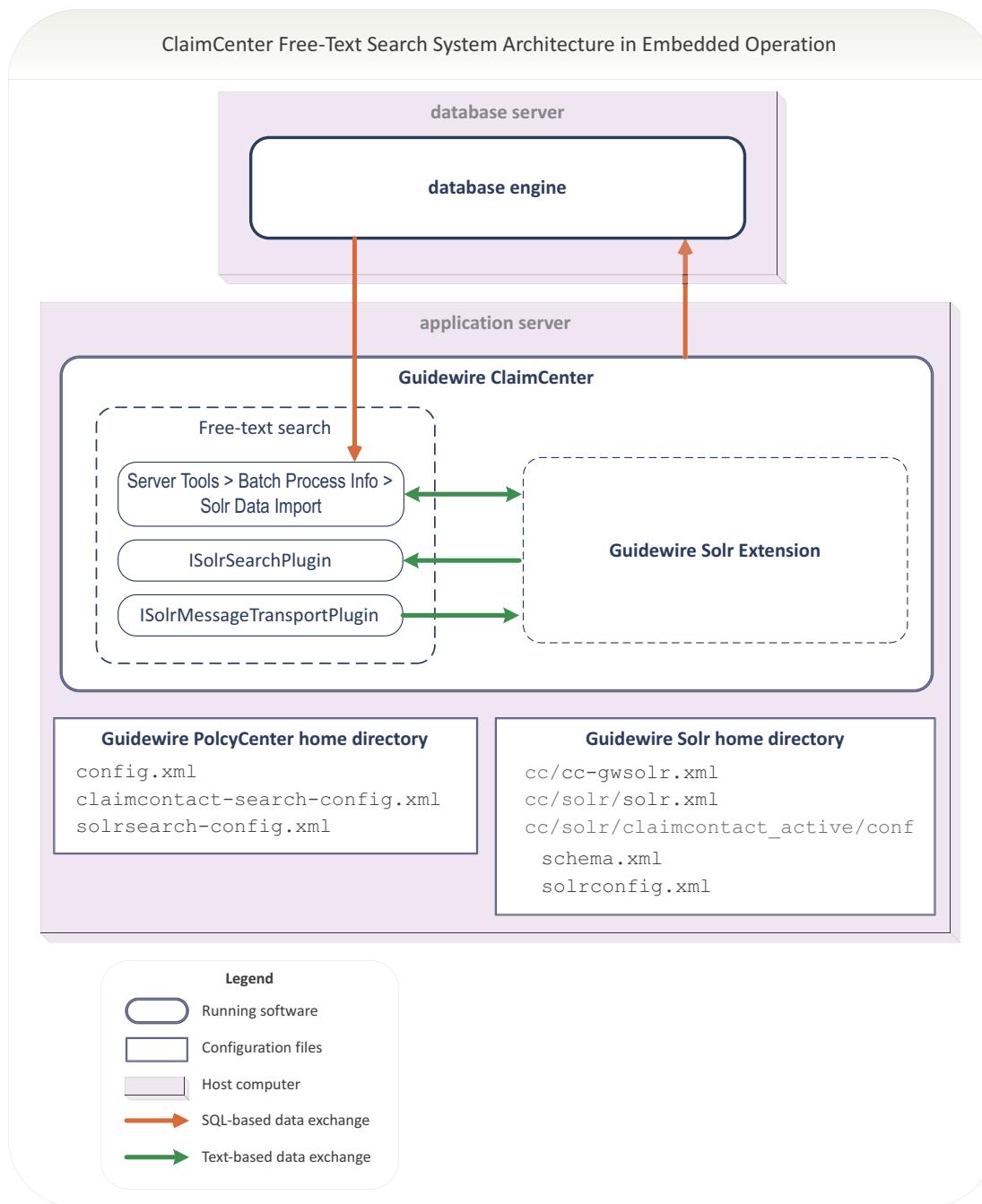
Free-text Search System Architecture in a Production Environment

The following diagram illustrates the system architecture for free-text search if you run ClaimCenter 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 ClaimCenter in a development environment. In a development environment, you can configure free-text search of embedded operation.



With embedded operation, the Solr Data Import batch process on the **Batch Process** page of the **Internal Tools** tab is available as the alternative to the free-text batch load command.

Free-text Search Configuration Parameters and Files

The free-text feature of ClaimCenter depends on configuration parameters and files in various locations.

Configuration Parameters and Files for Free-text Search in ClaimCenter

The following parameters and files help enable and configure free-text search in ClaimCenter:

- `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`.
- `claimcontact-search-config.xml` – Provides detailed configuration of the fields that free-text search extracts from the ClaimCenter database and sends to the full-text search database for indexing and searching.
- `solrserver-config.xml` – Configures how ClaimCenter 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 74.
- For details about the `solrserver-config.xml` configuration file, see “Configuring the Guidewire Solr Extension for Integration with ClaimCenter” on page 364.

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 ClaimCenter sends for indexing and how the Guidewire Solr Extension responds to search requests from ClaimCenter.

- `cc-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 Guidewire Solr Extension for Integration with ClaimCenter” on page 364
- “Configuring Free-text Search for Indexing and Searching” on page 367

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 ClaimCenter 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.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.xml` file to use.

See also

- “Configuring the Free-text Batch Load Command” on page 368

Enabling Free-text Search in ClaimCenter

Full-text search is disabled in the base configuration of ClaimCenter. 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 ClaimCenter with:

- The configuration parameter `FreeTextSearchEnabled` in `config.xml`
- 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 ClaimCenter 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 ClaimCenter off and on temporarily.

To enable free-text search in ClaimCenter

1. Follow the instructions for “Free-text Search Setup” on page 85 in the *Installation Guide*.
2. Start ClaimCenter Studio.
3. Navigate to Other Resources → `config.xml` and set `FreeTextSearchEnabled` to `true`.
4. Start the ClaimCenter 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:

```
***** CCSolrMessageTransportPlugin 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 cc
```

See also

- To set up free-text search for external or embedded operation, see “Free-text Search Setup” on page 85 in the *Installation Guide*.
- To learn more about the plugins and message destination, see “Free-text Search Integration” on page 595 in the *Integration Guide*

Configuring the Guidewire Solr Extension for Integration with ClaimCenter

The following configuration files are important when you configure the Guidewire Solr Extension for free-text search.

- `cc-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 ClaimCenter, the only supported core is for claim contacts.
- `solrserver-config.xml` – Configures how ClaimCenter 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 ClaimCenter 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 `cc-gwsolr.xml` and `solr.xml` configuration files, see “Free-text Search Setup” on page 85 in the *Installation Guide*.

Configuring Connections with the Guidewire Solr Extension

You configure instances of the Guidewire Solr Extension and how ClaimCenter 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 claim contact data, a `<document>` element associates that data type with an instance of the Guidewire Solr Extension. For example:

```
<document name="claimcontact" 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 ClaimCenter 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 ClaimCenter.
- `http` – The Guidewire Solr Extension operates externally from ClaimCenter, as a single server.
- `cloud` – The Guidewire Solr Extension operates externally from ClaimCenter, 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 ClaimCenter 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="claimcontact" archive="false" servername="embedded"/>
```

The `name` attribute lets you bind document types, such as polices, to a server that has cores for them. A `<document>` element must never reference a `<solrserver>` element of type `embedded` if you run the ClaimCenter application in production mode. If you do so, ClaimCenter 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 `cc-gwsolr.zip` file as `solrroot`. In a typical development environment, the home directory is on the same host as the one that hosts your ClaimCenter 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 366

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 ClaimCenter. 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 polices, 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 ClaimCenter application. If you run the Guidewire Solr Extension on the same machine that runs ClaimCenter, 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 a 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"/>
</solrserver>
```

Specify timeout intervals in milliseconds. The `connectiontimeout` parameter specifies how long ClaimCenter waits for the Guidewire Solr Extension to respond to a connection request. The `readtimeout` parameter specifies how long ClaimCenter 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 367

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 ClaimCenter/gwsolr/cc-gwsolr.zip to the solrroot directory every time you start the ClaimCenter 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 ClaimCenter application.

To use provisioning successfully

1. Always modify the free-text configuration files in Studio.
2. Stop the ClaimCenter application, if it is running.
3. Open a command prompt to PolicyCenter/bin and run the following command:
gwpc solr
The command rebuilds the cc-gwsolr.zip file in PolicyCenter/solr.
4. Start the ClaimCenter application.

Free-text search deploys the contents of cc-gwsolr.zip to the directory specified by solrroot. Files already in solrroot are overwritten by new files from cc-gwsolr.zip.

See also

- “Configuring the Guidewire Solr Extension for Embedded Operation” on page 365

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 claim contacts.

- schema.xml – Defines search fields as known in the Guidewire Solr Extension.
- claimcontact-search-config.xml – Defines the mapping between ClaimCenter 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 claim contact 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 85 in the *Installation Guide*
- “Modifying Free-text Search for Additional Fields” on page 369

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/cc/solr/claimcontact_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.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 ClaimCenter 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 94 in the *Installation Guide*.

Configuring the Search by Contact Screen for Free-text Search

The following table provides implementation details for each field on the **Search → Claims → Search by Contact** screen. The columns contain the following information:

- Field** – The field on the **Search by Contact** screen.
- Entity** – The entity type of the object that corresponds to this field.
- Search type** – Exact or inexact depending upon the search type.

Use this information to help configure the **Search by Contact** screen for free-text search.

Field	Entity	Search type
Role	ClaimSearchCriteria.FreeTextNameSearchType	Exact
Name	ClaimSearchCriteria.NameCriteria.Name	Inexact
Phone	ClaimSearchCriteria.NameCriteria.Phone	Exact
Address	ClaimSearchCriteria.AddressCriteria.AddressLine1	Inexact
City	ClaimSearchCriteria.AddressCriteria.City	Inexact
State	ClaimSearchCriteria.AddressCriteria.State Used to filter search results.	Exact
Postal Code	ClaimSearchCriteria.AddressCriteria.PostalCode Used to filter search results.	Exact
Date	Multiple Date objects, depending on user selection. Used to filter search results.	Exact

Note: `ClaimSearchCriteria` is used to map input fields on the search screen, and `FreeTextClaimSearchResults` is used to map the results received from Solr.

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

Limits on the Number of Free-text Search Results Returned

The `ISolrSearchPlugin` plugin implementation limits the number of results that the Guidewire Solr Extension returns to ClaimCenter. 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 598 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 ClaimCenter displays the first page of results.

For advanced claim contact search, modify the `SolrClaimContactSearchPanelSet` page configuration file. View and edit the file in Studio by navigating to `Page Configuration (PCF) → search → SolrClaimContactSearchPanelSet`.

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.

Sequence of Steps for Adding a Field to Free-text Search

Configuration file	Description
ClaimCenter	
claimcontact-search-config.xml	Defines the mapping between ClaimCenter fields and Guidewire Solr Extension fields, and configures how the Guidewire Solr Extension indexes and searches its index documents. ClaimCenter/modules/configuration/config/search/claimcontact-search-config.xml
CCSolrSearchPlugin.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. ClaimCenter/modules/gsrc/gw/solr/CCSolrSearchPlugin.gs
Guidewire Solr Extension	
schema.xml	Defines the document schema structure for the Guidewire Solr Extension. http://wiki.apache.org/solr/SchemaXml /opt/gwsolr/cc/solr/claimcontact_active/conf/schema.xml
data-config.xml	Read only at startup, defines where to locate and how to interpret data from the batch load command. http://wiki.apache.org/solr/DataImportHandler#Configuration_in_data-config.xml-1 /opt/gwsolr/cc/solr/claimcontact_active/conf/data-config.xml
Free-text batch load command	
batchLoad.sh	The batch load command itself. /opt/gwsolr/cc/solr/claimcontact_active/conf/batchload.sh
batchload-config.xml	Contains database connection information and brand-specific native SQL for selecting data from the ClaimCenter relational database. /opt/gwsolr/cc/solr/claimcontact_active/conf/batchload-config.xml

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 370
- “Define a New Free-text Field in ClaimCenter” on page 371
- “Define a New Free-text Field in the Batch Load Command” on page 371

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/cc/solr/claimcontact_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 ClaimCenter

Open the following file to define a new free-text field in ClaimCenter.

```
ClaimCenter/modules/configuration/config/search/claimcontact-search-config.xml
```

In Studio, access the file from **configuration** → **Other Resources** → `claimcontact-search-config.xml`. ClaimCenter 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 `claimcontact-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.ClaimContactAddress.PostalCode"/>
</IndexField>
```

The definition specifies that values of `postalCode` fields in the index documents sent to the full-text engine come from addresses on claim contacts, 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 ClaimCenter 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 this:

```
SELECT DISTINCT
  ...
  ccaddr.postalcodeinternal AS postalCode
```

```
...
FROM cc_claimcontact AS cc
...
INNER JOIN cc_claim ccaddr
    ON ccaddr.branchid = cc.id
...
WHERE
...
AND (ccaddr.EffectiveDate IS NULL OR ccaddr.EffectiveDate &lt;= cc2.EditEffectiveDate )
AND (ccaddr.ExpirationDate IS NULL OR ccaddr.ExpirationDate &gt; cc2.EditEffectiveDate )
...
```

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>
  <CLAIMCONTACT>
    <!-- data for one claim contact -->
  </CLAIMCONTACT>
</CONTAINER_ELEM>
```

The entry needed in `data-config.xml` in order to include the `postalCode` in the index is:

```
<field column="postalCode" xpath="/CONTAINER_ELEM/CLAIMCONTACT/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.

Configuring Special Page Functions

This topic describes how to configure special functionality related to pages.

This topic includes:

- “Adding Print Capabilities” on page 373
- “Linking to a Specific Page: Using an EntryPoint PCF” on page 379
- “Linking to a Specific Page: Using an ExitPoint PCF” on page 382

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 ClaimCenter interface. This section explains the print capabilities and how to use them. It covers the following topics:

- Overview of the Print Functionality
- Types of Printing Configuration
- Working with a PrintOut Page

Overview of the Print Functionality

You can use the ClaimCenter printing functionality to print the data visible on a ClaimCenter screen. You can control both the screen’s output format and which screen objects are printed. Most commonly, pages print as PDF but ClaimCenter 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 ClaimCenter you can print:

- the entire organization tree
- object lists such as the **Activities** list on the Desktop
- object details such as detailed information about a contact

Guidewire also defines a special “print claim file” feature. You use this feature to print, with a single action, all of a claim’s information—or to choose which parts of the file to print. You can use this functionality as provided or configure it to suit your company’s requirements. See “Working with a PrintOut Page” on page 376 for details.

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 setting globally. For information on configuration parameters, see “Application Configuration Parameters” on page 33.

<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>	(Optional) 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>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>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 <http://www.w3.org/TR/REC-CSS2>.

Security Related to Printing

There are two system permissions related to printing—`lvprint` and `claimprint`. The `lvprint` permission gives you the ability to print the information that appears in a list view. The `claimprint` gives you the ability to print a claim.

Roles with <code>claimprint</code> permission	Roles with <code>lvprint</code> permission
• Adjuster	• Adjuster
• Claims Supervisor	• Claims Supervisor
• Manager	• Manager
• Clerical	• Superuser
• Superuser	• Integration Admin
• New Loss Processing Supervisor	• New Loss Processing Supervisor

Gosu API Methods for Printing

ClaimCenter has a Gosu API, `gw.api.wb.print`, that contains a number of methods. Guidewire recommends that you avoid using these APIs in your print configurations with the exception of `ListViewPrintOptionsPopupAction` and `PrintSettings`.

Types of Printing Configuration

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 ClaimCenter.

- *Location printing* that prints the current page in PDF.
- *List view printing* that prints a list in PDF or CSV format.
- *Customizable printing* that provides a list of print options.

This section discusses how to achieve the different kinds of print behaviors.

Location Printing

Adding the ability to print the current location in PDF is the simplest functionality to configure. You simply add a `PrintToolbarButton` element to a `Screen` element. For example, the following line in the `DashboardClaimCount` PCF file creates a Print button in the toolbar:

```
<Page id="DashboardClaimCount"
      title="displaykey.Java.Dashboard.Title(displaykey.Java.Dashboard.ClaimCount.Title)"
      canVisit="perm.User.viewedbclaimcounts">
  <LocationEntryPoint signature="DashboardClaimCount(GroupId : web.dashboard.DashboardTreeGroupInfo)" />
  ...
  <Toolbar>
    <PrintToolbarButton id="print" label="displaykey.Button.Print"
      available="perm.User.printlistviews"/>
    <ToolbarDivider/>
  ...
</Screen>
</Page>
```

This causes the current location, `DashboardClaimCount` to print. You can also refer to another location in the print bar by providing a `locationRef`.

List View Printing

List view printing allows you to 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` method. The following example illustrates a list view printing method:

```
<ToolbarButton label="displaykey.Java.ListView.Print" id="PrintButton"
  action="web.print.ListViewPrintOptionPopupAction.printListViewWithOptions('MyListView')"/>
```

If you choose to print in CSV format, then you can also choose which columns to print. For example, you can use list view printing to print the `Desktop Activities` page.

Customizable Printing

Customizable printing allows you to create a print options page that controls exactly what ClaimCenter prints. This printing feature uses a special PCF page containing a `PrintOut` element that is comparable to a `Screen` element. Using a `PrintOut` page, you can:

- Print an entire page or select parts of a page to print.
- Print a list view or a list view and its item details.
- Apply a filter to a list view before printing it.

ClaimCenter uses customizable printing to print the claim file.

You can only print a list view's items in detail using customizable print. Not all list views are eligible for printing details. Only list views that are screen panels are eligible for detail printing—for example a `CardPanel` element or the `ListViewPanel` in Claim `Workplan`. You cannot print the detail for lists embedded in detail views.

Working with a PrintOut Page

IMPORTANT This section explains the configuration of the ClaimPrintOut PCF page. Guidewire recommends that you refrain from creating your own PrintOut pages. Instead, Guidewire recommends that you modify the existing ClaimPrintOut PCF page, limiting yourself to adding support for your subtypes and extensions.

A PrintOut page is always interactive. The page displays one or more groups of radio buttons and check boxes that control what ClaimCenter actually prints. To use a PrintOut page, you define print elements in a single PCF file. For organizational purposes, the file name usually contains the word print or is stored in a print directory. ClaimCenter contains a single PrintOut page, ClaimPrintOut.pcf.

To call this PrintOut file, you trigger an action from a menu item or button that calls the page:

```
<MenuItem label="displaykey.Java.ClaimMenu.PrintClaim"
          action="ClaimPrintout.push(Claim)" id="ClaimMenuActions_Print"/>
```

The page takes as the current claim as a variable and offers various options to print the claim.

PrintOut Attributes and Subelements

A PrintOut page has the following attributes and subelements (required elements and attributes are in bold):

```
<PrintOut id="string" canVisit="expression" desc="string" parent="string" title="string">
  <Code/>
  <LocationEntryPoint canVisit="expression" parent="string" signature="string" title="string" />
  <Variable initialValue="expression" name="string" recalculateOnRefresh="boolean" type="string" />
  <Verbatim desc="string" hideIfEditable="boolean" hideIfReadOnly="boolean"
            id="string" label="string" labelStyleClass="string" visible="expression"
            warning="boolean" />
  <PrintGroup choosable="expression" customizable="boolean" id="string" label="string">
    <PrintSection choosable="expression" id="string" label="string">
      <PrintOption choosable="expression" id="string" label="string">
        <PrintOptionLocation filter="expression" listViewRef="string" locationRef="string"
                              printable="string">
          <PrintDetail locationRef="string" mode="string" symbolName="string" symbolType="string" />
        </PrintOptionLocation>
      </PrintOption>
    </PrintSection>
  </PrintGroup>
  <PrintLocation id="string" label="expression" printable="expression">
    <PrintLocationDetail baseLocation="string" filter="expression" listViewRef="string"
                          locationRef="string" printable="expression" symbolName="string" symbolType="string" />
  </PrintLocation>
  <PrintOutButton action="action" label="expression" shortcut="string" />
</PrintOut>
```

Each PrintOut page must contain either one PrintGroup or one PrintLocation. The following PrintOut configuration subelements are specific to customizable printing:

Element	Description
PrintDetail	Instructions on how to print elements if you elect to print details.
PrintGroup	Defines a set of pages to print. This element contains one or more PrintSection elements. ClaimCenter represents each PrintSection element in the interface with a radio button. You can customize a PrintGroup. See “Customizable Printing” on page 375 for more information on using a customizable group.
PrintLocation	Specifies a specific location to print. This element takes one or more PrintLocationDetail elements.
PrintLocationDetail	Instructions on how to print elements if you elect to print details. ClaimCenter represents each print element in the interface with a radio button.
PrintOption	A set of locations that are printed together. Each PrintOption contains one more PrintOptionLocation subelements.
PrintOptionLocation	Specifies a page (location) to print any time that you select a PrintGroup. This element can contain one PrintDetail.

Element	Description
PrintOutButton	Adds a button to a PrintOut page. This element triggers printing if you select print options, or if you cancel and close the page.
PrintSection	Represents a print selection on a PrintOut page. Each PrintSection must contain one or more PrintOption elements.

With the exception of the PrintDetail element, all of the PrintOut subelements specify a `printable` attribute. This attribute takes a boolean expression that determines if the print option is visible. If the expression evaluates to true, the option appears.

Printing All the Claim's Pages with and without Detail

The **All pages excluding details** option is represented in the file by a single `PrintGroup` element. A `PrintGroup` defines a set of pages to print. Each set is contained in a `PrintSection` element. The individual pages appear as `PrintOptionLocation` elements within a `PrintOption`. The following section illustrates the definition for the claim workplan pages:

```
<PrintGroup id="AllClaimPagesWithoutDetails"
    label="displaykey.Java.PrintClaimOptionsForm.Label.AllClaimPagesWithoutDetails">
    ...
    <PrintSection id="WorkplanSection" label="displaykey.Java.PrintClaimOptionsForm.Label.Workplan"
        printable="perm.System.viewworkplan">
        <PrintOption id="WorkplanSummaryOption"
            label="displaykey.Java.PrintClaimOptionsForm.Options.Workplan.Summary">
            <PrintOptionLocation locationRef="ClaimWorkplan(Claim)" />
        </PrintOption>
    </PrintSection>
    ...
</PrintGroup>
```

The **All Pages including details and FNOL snapshot** option is defined in a second `PrintGroup` that allows you to print the details of each page. This is accomplished through the addition of the optional `PrintDetail` element.

```
<PrintSection id="WorkplanSection" label="displaykey.Java.PrintClaimOptionsForm.Label.Workplan"
    printable="perm.System.viewworkplan">
    <PrintOption id="WorkplanDetailsOption"
        label="displaykey.Java.PrintClaimOptionsForm.Options.Workplan.Details">
        <PrintOptionLocation locationRef="ClaimWorkplan(Claim)" />
        <PrintDetail symbolName="Activity" locationRef="ActivityDetailPrint(Activity)"
            symbolType="Activity"/>
    </PrintOptionLocation>
</PrintOption>
</PrintSection>
```

You can only configure one list view on an individual page to print. (In other words, you can configure only one list view to print for each screen.) The `locationRef` attribute specifies a page that takes the specified `symbolName` and `symbolType` and processes them for printing. The page in this case is the pcf/shared/printing/ActivityDetailPrint.pcf page. This page takes the `ClaimPrintOut.pcf` as a parent.

The file that defines the action list view determines what you specify for the `symbolName` and `symbolType` attributes. In this case, that file is the `pcf/claim/workplan/WorkplanLV.pcf`. This file populates itself from an array of `Activity` elements:

```
<ListViewPanel id="WorkplanLV">
    ...
    <Require name="ActivityList" type="Activity[]"/>
    ...
    <RowIterator elementName="Activity" editable="false"
        value="ActivityList" hasCheckboxes="true"
        hideCheckboxesIfReadOnly="false">
    ...

```

This page requires an `Activity` type and elements of this type are named `Activity`. In the print out page, the `symbolType` originates from the `type` value in the list view and the `symbolName` from the `elementName` value.

Printing the Current Page with and without Detail

To print the current page, the page on which a user initiated the print action, you use `PrintLocation` elements. To print the entire page without details, you simply supply a single `PrintLocation`:

```
<PrintLocation id="CurrentClaimfilePagePrint"
    label="displaykey.Java.PrintClaimOptionsForm.Label.ThisPageWithoutDetails"/>
```

To print the detail on a page, you need to account for the details on each possible page — based on the location of the print action. In the case of a claim, the action appears in the side menu and so can appear from any page in the claim. For example:

```
<PrintLocation id="CurrentClaimfilePagePrintWithDetails"
    label="displaykey.Java.PrintClaimOptionsForm.Label.ThisPageWithDetails">

    <PrintLocationDetail baseLocation="ClaimWorkplan" symbolName="Activity"
        locationRef="ActivityDetailPrint(Activity)" symbolType="Activity"/>

    <PrintLocationDetail baseLocation="ClaimAssociations"
        locationRef="ClaimAssociationDetail(Claim, ClaimAssociation)"
        symbolName="ClaimAssociation" symbolType="ClaimAssociation"/>
    ...
    <PrintLocationDetail baseLocation="ClaimMatters" symbolName="Matter"
        locationRef="MatterDetailPage(Claim, Matter)" symbolType="Matter"/>
</PrintLocation>
```

The specification of the `locationRef`, `symbolName`, and `symbolType` all use the same principles as the `PrintDetail` element.

Configuring a Customizable PrintGroup

A customizable `PrintGroup` appears as regular radio buttons. If a user selects the button, ClaimCenter displays a list of checkbox and drop-down options. `PrintSection` subelements appear as check boxes. `PrintOption` subelements display as accompanying drop-down menus. For example, the following

```
<PrintGroup id="Custom" label="displaykey.Java.Printout.Custom" customizable="true">
    ...
    <PrintSection id="WorkplanSection" label="displaykey.Java.PrintClaimOptionsForm.Label.Workplan"
        printable="perm.System.viewworkplan">
        <PrintOption id="WorkplanSummaryOption"
            label="displaykey.Java.PrintClaimOptionsForm.Options.Workplan.Summary">
                <PrintOptionLocation locationRef="ClaimWorkplan(Claim)"/>
            </PrintOption>
        <PrintOption id="WorkplanDetailsOption"
            label="displaykey.Java.PrintClaimOptionsForm.Options.Workplan.Details">
                <PrintOptionLocation locationRef="ClaimWorkplan(Claim)">
                    <PrintDetail symbolName="Activity" locationRef="ActivityDetailPrint(Activity)"
                        symbolType="Activity"/>
                </PrintOptionLocation>
            </PrintOption>
        </PrintSection>
    ...
</PrintGroup>
```

This group gives you the option to print each section of the claim file either with details or without.

Filtering a List View

You can apply a filter to a list view to control what is or is not printed out. To do this, you would specify a `filter` attribute on either the `PrintOptionLocation` or the `PrintLocationDetail` element — which ever applies. The following example illustrates the use of a filter:

```
<PrintSection id="HistorySection" label="displaykey.Java.PrintClaimOptionsForm.Label.History"
    choosable="perm.System.viewclaimhistory">
    <PrintOption id="HistoryAllOption" label="displaykey.Java.HistoryFilter.All">
        <PrintOptionLocation locationRef="ClaimHistory(Claim)" listViewRef="HistoryLV"
            filter="FilterSet.AllFilter"/>
    </PrintOption>
    <PrintOption id="HistoryAssignmentOption" label="displaykey.Java.HistoryFilter.Assignment">
        <PrintOptionLocation locationRef="ClaimHistory(Claim)" listViewRef="HistoryLV"
            filter="FilterSet.AssignmentFilter"/>
    </PrintOption>
    <PrintOption id="HistoryViewingOption" label="displaykey.Java.HistoryFilter.Viewing">
        <PrintOptionLocation locationRef="ClaimHistory(Claim)" listViewRef="HistoryLV"
            filter="FilterSet.ViewingFilter"/>
    </PrintOption>
</PrintSection>
```

```
        filter="FilterSet.ViewingFilter"/>
    </PrintOption>
</PrintSection>
```

The print section has three filters all applied to the HistoryLV.pcf list view. You configure your list view to use any valid and relevant list view filter — it would not make sense to use an activity filter on an exposure list.

Overriding the Print Settings in a File

Any of the global print settings you set globally in the config.xml file, you can override within an individual PCF file. You do this using a Gosu API (gw.api.web.print.PrintSettings). The following example shows how you would override the global font size in a dialog:

```
<Page id="ClaimExposures" title="displaykey.Web.Claim.Exposures"
    canEdit="false" canVisit="Claim.ExposureListChangeable and perm.Claim.view(Claim)
    and perm.System.viewexposures">
...
    <Variable name="PrintTargetLV" initialValue=""ExposuresLV""/>
    <Variable name="PrintSettings" type="print.PrintSettings" initialValue="createPrintSettings()"/>
    <Variable name="PrintClaimNumber" type="String"
        initialValue="displaykey.Web.PrintOut.ClaimNumber(Claim.ClaimNumber)"/>
...
    <Toolbar>
...
    <ToolbarButton label="displaykey.Java.ListView.Print" id="ClaimExposures_Print"
        shortcut="N"
        available="perm.User.printlistviews and perm.Claim.print(Claim)"
        action="print.ListViewPrintOptionPopupAction.printListViewWithOptions(PrintTargetLV,
        PrintSettings)"/>
...
    <Code>
        function createPrintSettings() : print.PrintSettings {
            var newPrintSettings = new print.PrintSettings();
            var claimNumberLabel = displaykey.Web.PrintOut.ClaimNumber(Claim.ClaimNumber);
            newPrintSettings.setHeaderLabel(claimNumberLabel);
            newPrintSettings.setFontSize("12px");
            return newPrintSettings;
        }
    </Code>
</Page>
```

These print settings only apply if printing the page by itself — the Print button action sets them before printing. If you are printing this page as part of a claim file, the settings do not apply.

Troubleshooting Print Configurations

If you run into problems with the display of a printed page, for example elements overlap, you can try the following:

- Adjust the print widths of the columns using the `PrintSettings` Gosu API.
- Hide unwanted columns and print only the necessary columns. Unlike a screen, the printed page is limited by the available width.
- Change the font size of the printed page using the `PrintSettings` Gosu API in a list view.

Linking to a Specific Page: Using an EntryPoint PCF

It is possible to connect directly to ClaimCenter using a URL that leads to a specific ClaimCenter page. You can define your own links or entry points. Thus, if the ClaimCenter server receives a connect request from an external source and the request has both the correct format and parameters, ClaimCenter serves the requested page.

In the base configuration, ClaimCenter provides a number of `EntryPoint` PCF examples. You can find these in the following location in Studio:

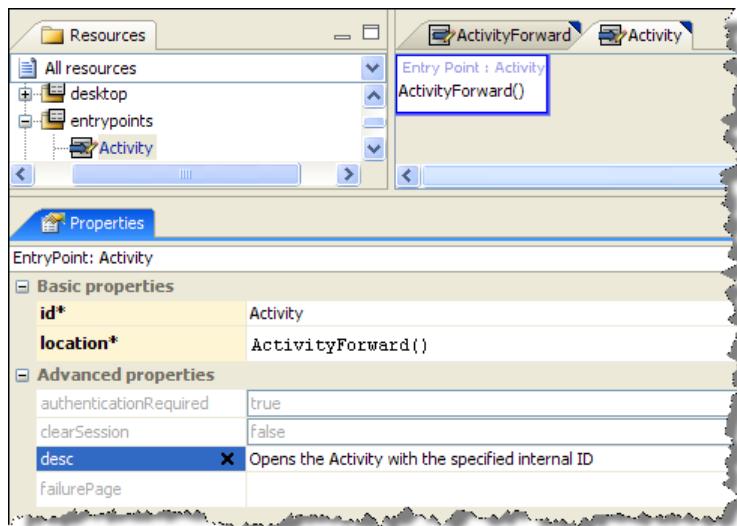
`configuration → config → Page Configuration → pcf → entrypoints`

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 ClaimCenter server receives a connection request with a specific entry point, ClaimCenter responds by serving the page based on the entry point configuration.

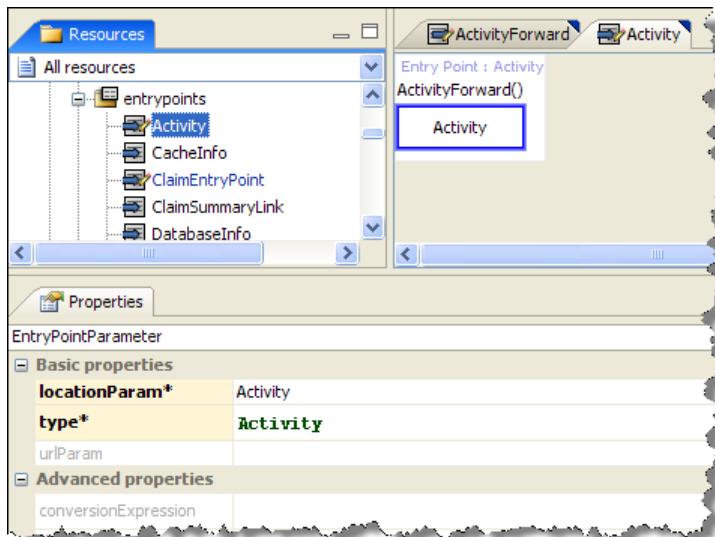
To implement this functionality, you must create an **EntryPoint** PCF (in the `entrypoints` folder). The following graphic illustrates an **EntryPoint** PCF.



The **EntryPoint** PCF contains the following parameters:

<code>authenticationRequired</code>	Specifies that ClaimCenter must authenticate the user before the user can access the URL. If true, ClaimCenter requires that the user already be authenticated to enter. If the user is not already logged in, ClaimCenter 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 ClaimCenter 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 ClaimCenter 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, ClaimCenter, 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 381 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 ClaimCenter 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 <code>LocationParameter</code> on the <code>EntryPoint</code> target location that this parameter sets.
<code>optional</code>	Specifies whether the parameter is optional. If set to <code>true</code> , ClaimCenter does not require this parameter.
<code>type</code>	Required. Specifies what type to cast the incoming parameter into, such as <code>String</code> or <code>Integer</code> .
<code>urlParam</code>	The name of the parameter passed with the URL. For example, if the <code>urlParam</code> is <code>Activity</code> and the entry point URI is <code>ActivityDetail</code> , 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 `ClaimForward` in ClaimCenter Studio at `pcf→claim→ClaimForward`.

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 ClaimCenter 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 ClaimCenter screen.

In the base configuration, ClaimCenter 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 ClaimCenter 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 ClaimCenter. 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 ClaimCenter. 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 ClaimCenter 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 ClaimCenter 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 ClaimCenter data model, which would require a server restart.

2. Log into ClaimCenter 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. ClaimCenter presents a success message after it reloads the PCF files from the local file system.
5. Log into ClaimCenter 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 ClaimCenter opens a popup window and loads the URL that you set on the `constructMyURL` function. If you followed the steps of this example exactly, ClaimCenter 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 ClaimCenter” on page 387
- “Workflow in Guidewire Studio” on page 387
- “Understanding Workflow Steps” on page 389
- “Using the Workflow Right-Click Menu” on page 390
- “Using Search with Workflow” on page 390

Workflow in Guidewire ClaimCenter

Guidewire ClaimCenter uses workflow primarily to support MetroReport integration. Guidewire defines and stores each base configuration workflow process as a separate file in the following directory:

```
.../modules/cc/config/workflow
```

Each file name corresponds to the workflow process that it defines (for example, `MetroReport.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 information on workflow structure and design, see “Guidewire Workflow” on page 393.

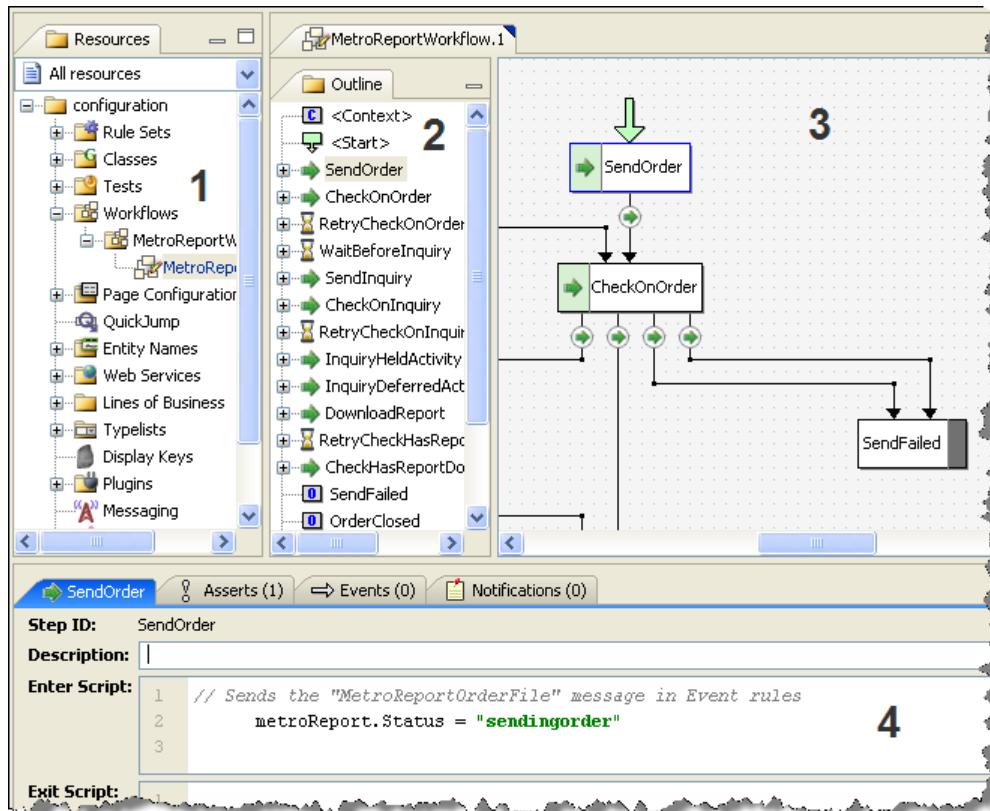
Workflow in Guidewire Studio

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 ClaimCenter base configuration, Guidewire defines a **MetroReportWorkflow** 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 399
<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 399
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 401
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 402
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 403
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 404
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 407
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 408
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 409
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 405
<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 399

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. ClaimCenter 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 398.

Type	Workflow contains	Icon	Step	Description
AutoStep	Zero, one, or more		Step1	Step that ClaimCenter guarantees to finish immediately. See “AutoStep” on page 401.
ManualStep	Zero, one, or more		Step2	Step that waits for an external TRIGGER to occur or a TIMEOUT to pass. See “ManualStep” on page 404.
ActivityStep	Zero, one, or more		Step3	Step that waits for one or more activities to complete before continuing. See “ActivityStep” on page 403.
MessageStep	Zero, one, or more		Step4	Special-purpose step designed to support messaging-based integrations. See “MessageStep” on page 402.
Outcome	One or more		Outcome	Special final step that has no branches leading out of it. See “Outcome” on page 405.

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 53 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 ClaimCenter workflow. Workflow is the Guidewire generic component for executing custom business processes asynchronously.

This topic includes:

- “Understanding Workflow” on page 394
- “Workflow Structural Elements” on page 398
- “Common Step Elements” on page 399
- “Basic Workflow Steps” on page 401
- “Step Branches” on page 406
- “Creating New Workflows” on page 411
- “Instantiating a Workflow” on page 414
- “The Workflow Engine” on page 417
- “Workflow Subflows” on page 420
- “Workflow Administration” on page 421
- “Workflow Debugging and Logging” on page 423

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.

Note: 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. ClaimCenter creates a workflow instance in response to a specific need to perform a task or function, usually asynchronously. For example, in the base configuration, ClaimCenter provides a ready-to-use integration to the Metropolitan Reporting Bureau (www.metroreporting.com) that it bases on workflow. (You use this workflow as an aid in obtaining police reports of accidents.)

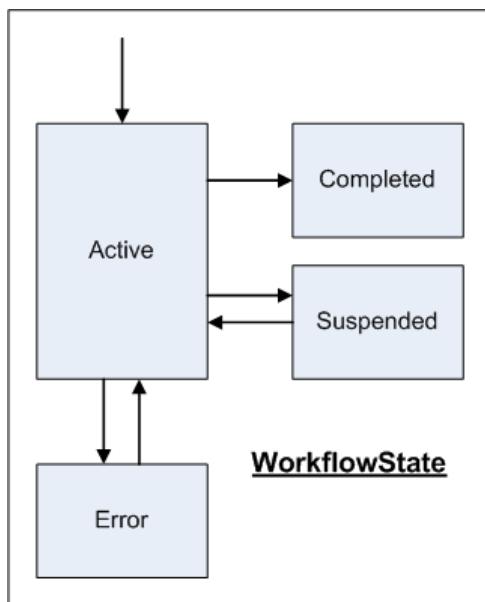
The newly created instance takes the form of a database entity called `Workflow`. (For more information on the `Workflow` entity, consult the ClaimCenter *Data Dictionary*.) Because ClaimCenter creates the `Workflow` entity in a bundle with other changes to its associated business data, ClaimCenter does nothing with the workflow until it commits the workflow. ClaimCenter 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 ClaimCenter **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 “Distributable Work Queues” on page 124 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 – ClaimCenter retries a Failed work item up to the maximum retry limit.
- Checkedout – ClaimCenter 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 Distributable Work Queues” on page 128 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 387.

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 “claim” actually inserts “Workflow.Claim”.
<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 ClaimCenter 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 387.

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.

Workflow Versioning

After you create a workflow script and make it active, it can create hundreds or even thousands of working instances in the ClaimCenter 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 ClaimCenter 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 414.

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. ClaimCenter 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:

- ClaimCenter records the current step of an active workflow in the database. Each change to the basic structure of a workflow requires a new version.
- ClaimCenter records the branch that an active workflow selects in the database. A change to the Branch ID requires a new version.

- ClaimCenter records the activity associated with an Activity step in the database. A change to an Activity definition requires a new version.
- ClaimCenter records the trigger activity that occurs in an active workflow in the database. A removal of a trigger requires a new workflow version.
- ClaimCenter 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 `TIMOUT` step. ClaimCenter does record the wake-up time (for a `TIMOUT` 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 53 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 53 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 claim actually inserts Workflow.Claim.

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 **ClaimCenter 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, ClaimCenter 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 399 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 299 in the <i>Integration Guide</i>. • For what constitutes a valid event name, specifically see “List of Messaging Events in ClaimCenter” on page 320 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 ClaimCenter sends out, but which does not block the workflow from continuing. ClaimCenter 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 403 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 ClaimCenter.
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 ClaimCenter 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 `GO` branch. (It can have more than one, but it must have at least one.) Each `GO` 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 `GO` branches (from top to bottom in the XML code):

- It picks the first `GO` 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 399 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), ClaimCenter stores an optional response string (supplied with the ack) on the message in the `Response` field. ClaimCenter 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 399 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. ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 ClaimCenter.
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 ClaimCenter. 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 409 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.

Outcome

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 407), 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 ClaimCenter 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 ClaimCenter interface. Typically, in ClaimCenter, you invoke a trigger though 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 418 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 ClaimCenter *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 ClaimCenter 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.

Field	Description
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 ree.
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 411. 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:

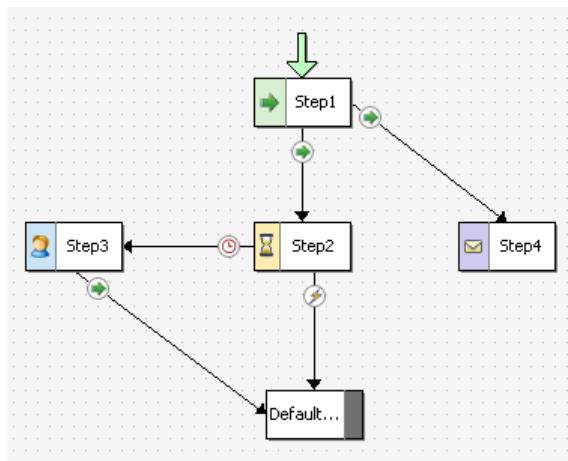
- The outline view contains the few required workflow elements.
- The layout view contains a default outcome (DefaultOutcome).

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 397.

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 ClaimCenter Workflow console

See Also

- See “Workflow Debugging and Logging” on page 423 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 ClaimCenter `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.

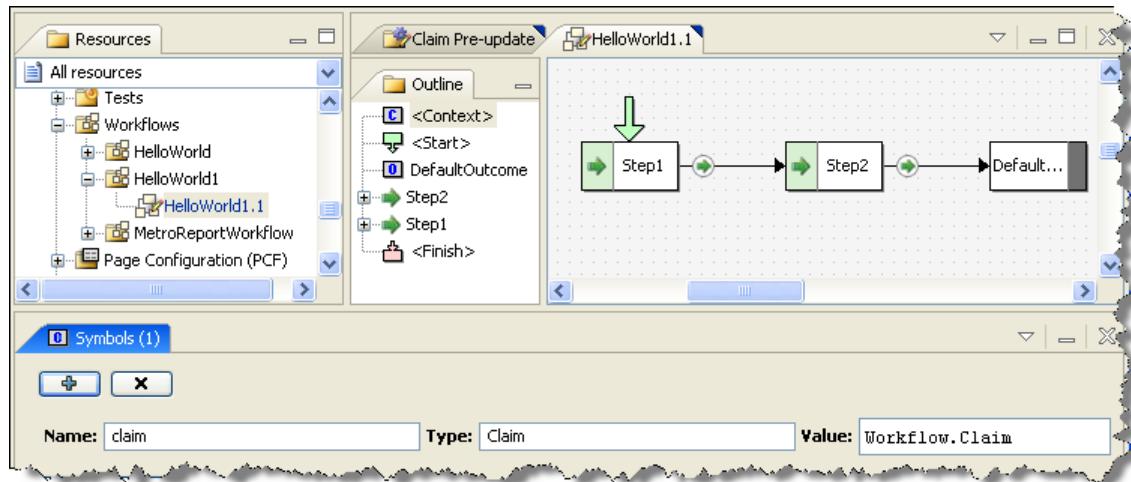
To cause updates to happen, the example instantiates the workflow from an `Edit` screen in ClaimCenter. It then calls a Claim Pre-Update Rule in Studio.

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

ClaimCenter uses a Distributed Worker 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. ClaimCenter creates a `WorkflowWorkItem` instance to advance the workflow.
2. The distributed 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, ClaimCenter 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 414. 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 ClaimCenter 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` typelist.

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 421.)
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 ClaimCenter 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 53 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	If any exception occurs during <i>synchronous</i> execution, even after the workflow has gone through several steps, ClaimCenter 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). <ul style="list-style-type: none"> • If you start the workflow or invoke the trigger from the ClaimCenter interface, ClaimCenter displays the exception in the interface. • If some other code started the workflow, that code receives the exception.
Asynchronous	If any exception occurs during <i>asynchronous</i> execution (as it executes in the background), ClaimCenter logs the exception and rolls back the bundle, in a similar manner to the synchronous case. <p>ClaimCenter then handles workflow retries in the standard way through the distributed worker. ClaimCenter 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, ClaimCenter 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 Administration tab within ClaimCenter. 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"> • entering the step • following the branch • or, attempting to perform whatever it was doing at the time the exception occurred <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 ClaimCenter 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 55 in the *Globalization Guide*

Workflow Administration

You can administer workflow in any of the following ways:

- Through the ClaimCenter **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 ClaimCenter **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 ClaimCenter 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, ClaimCenter 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, ClaimCenter 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, ClaimCenter suspends the selected workflows. This action is appropriate for all workflow and steps. However, ClaimCenter 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, ClaimCenter 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

ClaimCenter 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.

See also

- “Managing Workflows” on page 484 in the *Application Guide*

Workflow Debugging and Logging

Debugging a workflow is a more challenging task than debugging the standard ClaimCenter 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 ClaimCenter. (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 ClaimCenter 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
```


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 425
- “Pattern Types and Categories” on page 426
- “Using Activity Patterns in Gosu” on page 427
- “Calculating Activity Due Dates” on page 427
- “Defining the Business Calendar” on page 428
- “Configuring Activity Patterns” on page 429
- “Using Activity Patterns with Documents and Emails” on page 431
- “Localizing Activity Patterns” on page 431

What is an Activity Pattern?

Activity patterns standardize the way that Guidewire ClaimCenter creates activities. Activity patterns describe the kinds of activities that people perform while handling claims within an organization. For example, obtaining a statement from a witness 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 claim, ClaimCenter 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 claims business processes from the **Administration** tab in ClaimCenter. It is possible to create activities from activity patterns in different ways:

- You can manually create activities in ClaimCenter.

- A business rule or some other Gosu code create activities as part of generating workplans or while responding to escalations, claim exceptions, or other events.
- ClaimCenter 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 the **Claim** or **Exposure** page.

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 ClaimCenter assigns an activity. Instead, activity assignment methods in the assignment rules or in Gosu expressions control how ClaimCenter assigns an activity. Using the pattern name, the assignment methods determine to whom to assign the activity.

Pattern Types and Categories

ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 ClaimCenter 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 274.

In the base configuration, Guidewire ClaimCenter 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 ClaimCenter interface. You use them only within Gosu and ClaimCenter 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 ClaimCenter **New Activity** menu. The activity pattern categories appear only within the ClaimCenter 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 ClaimCenter interface.

ClaimCenter 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 170 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 38 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 ClaimCenter calculates the due date for an activity. ClaimCenter can calculate a target due date in hours or days. ClaimCenter 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. For example, suppose that you want to schedule a vehicle inspection to be done within five days of the (company service-level target). You specify this using the **Target days** or **Target hours** value.
- *What is the starting point?* What point in time does ClaimCenter use as the start point in calculating the target date? For example, is the goal to perform a vehicle inspection within 5 days of the loss date or the claim's first notice? You specify this using the **Target start point** field.
- *What days to count?* ClaimCenter can count calendar days or only business days. You specify this with the **Include these days** field.

ClaimCenter 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. ClaimCenter 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**, ClaimCenter uses 0 for both. A target date is optional for activities. For example, suppose that there is simply no reason to set an target date for adjuster self-reminders.

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.) ClaimCenter 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, ClaimCenter 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.

ClaimCenter 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, ClaimCenter uses 0 (zero) for both. An escalation date, like a target date, is optional for activities.

Defining the Business Calendar

Due dates within ClaimCenter depend on an understanding of the business calendar as defined by your company. For example, if something is due in five business days, exactly which days does this include? Does your company operate seven days a week or do you consider only Monday through Friday to be business days? Which days are company holidays? Another key concept in the business calendar is understanding how your company defines the point at which a week begins or ends. Is Friday the last day of the week or is Sunday?

You manage the business calendar through the **Administration → Holidays** screen within ClaimCenter.

See also

- “Holidays and Business Weeks” on page 259 in the *Application Guide*

Configuring Activity Patterns

ClaimCenter 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 ClaimCenter Administration tab. You can access the `activity-patterns.csv` file through Guidewire Studio by navigating to the Resources → Other Resources → import folder.

IMPORTANT Do not remove any internal (non-General type) activity patterns or change their type, category, or code values. Internal ClaimCenter application code requires them. You can change other fields associated with these types, however.

The `ActivityPattern` object contains a number of properties. The following list describes some of the more important properties:

Property	ClaimCenter field	Description
<code>ActivityClass</code>	<code>Class</code>	Indicates whether the activity is a task or an event. A task has a due date. An event does not.
<code>AutomatedOnly</code>	<code>Automated only</code>	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">• If true, the activity pattern does not appear as a choice in ClaimCenter interface.• If you do not specify this value, the default is false. 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 ClaimCenter interface.
<code>Category</code>	<code>Category</code>	The category for grouping <code>ActivityPatterns</code> in the ClaimCenter interface (in a drop-down list).
<code>ClaimLossType</code>	<code>Claim loss type</code>	Describes the claim type for which the activity pattern is relevant. Valid options include the following: <ul style="list-style-type: none">• auto — Auto• property — Property• gl — General Liability• wc — Workers' Comp If not specified, the activity pattern is available for <i>all</i> types of claims.
<code>ClosedClaimAvlble</code>	<code>Available for closed claim</code>	A Boolean value that defines whether you can add the activity to a closed claims—meaning it is possible to perform the activity on a closed claim. If you do not specify a value, ClaimCenter uses a default of true.
<code>Code</code>	<code>Code</code>	Any text (<i>with no spaces</i>) that you can use to identify the pattern any time that you access the activity pattern in rules or Gosu code. You can only see this value through the Administration tab.
<code>Command</code>	<code>None</code>	<i>Do not use.</i> For Guidewire use only.
<code>Data-Set</code>	<code>None</code>	The value of the highest-numbered data set of which the imported object is a part. ClaimCenter 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.
<code>Description</code>	<code>Description</code>	Describes the expected outcome at the completion of this activity. It is visible only if you view the details of the activity. This value is optional.
<code>DocumentTemplate</code>	<code>Document Template</code>	Document template to display if you choose this activity. Enter the document template ID.
<code>EmailTemplate</code>	<code>Email Template</code>	Email template to display if you choose this activity. Enter the email template file name.
<code>EntityId</code>	<code>None</code>	<i>Required.</i> The unique public ID of the activity pattern.

Property	ClaimCenter field	Description
EscalationDays	Escalation days	The number of days from the <code>escalationstartpt</code> to set the Escalation Date for an activity. This value is optional.
EscalationHours	Escalation hours	The number of hours from the <code>escalationstartpt</code> to set the Escalation Date for an activity. This value is optional.
EscalationInclDays	Include these days	Specifies which days to include. You can set this <code>businessdays</code> or <code>elapsed</code> .
EscalationStartPt	Escalation start point	<p>The initial date used to calculate the target date. If you specify <code>escalationdays</code> or <code>escalationhours</code>, you need to specify this parameter. Otherwise, this parameter is optional.</p> <p>You can set this field to the following values:</p> <ul style="list-style-type: none"> • <code>activitycreation</code> — The activity's creation date. • <code>claimnotice</code> — The FNOL date which is the claim's "reported date." • <code>lossdate</code> — The date the accident or injury occurred
ExternallyOwned	Externally owned	A Boolean value indicating whether an external organization or user can own the activity or not.
Importance	Calendar importance	Specifies the default level of importance at which the activity appears on a calendar. You can set this value to the following values: <ul style="list-style-type: none"> • <code>top</code> • <code>high</code> • <code>medium</code> • <code>notoncalendar</code> • <code>low</code>
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 <code>true</code> .
Priority	Priority	Used to sort more important activities to the top of a list of work. You can set this property to the following values: <ul style="list-style-type: none"> • <code>urgent</code> • <code>high</code> • <code>normal</code> • <code>low</code>
Recurring	Recurring	<p>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 <code>true</code>.</p> <p>For example, a recurring <i>30 day diary</i> activity instructs the adjuster to check the claim every 30 days.</p>
ShortSubject	Short Subject	A brief description of the activity used on small areas of the ClaimCenter interface such as a calendar event entry.
Subject	Subject	A short text description of the activity that ClaimCenter shows in activity lists.
TargetDays	Target days	The number of days from the <code>targetstartpoint</code> to set the activity's Target Date. This value is optional.
TargetHours	Target hours	The number of hours from the <code>targetstartpoint</code> to set the activity's Target Date. This value is optional.
TargetIncludeDays	Include these 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"> • <code>elapsed</code>—the count all days • <code>businessdays</code>—as defined by the business calendar

Property	ClaimCenter field	Description
TargetStartPoint	Target start point	<p>The initial date used to calculate the target date. You need specify this value only if you specify <code>targetdays</code> or <code>targethours</code>. Otherwise, this value is optional.</p> <p>You can set this property to the following values:</p> <ul style="list-style-type: none"> • <code>activitycreation</code> — The activity's creation date. • <code>claimnotice</code> — The FNOL date which is the claim's "reported date." • <code>lossdate</code> — The date the accident or injury occurred.
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 ClaimCenter 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 ClaimCenter 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, ClaimCenter does the following:

- If you create a new activity from this activity pattern, ClaimCenter automatically populates any template field for which you specified a template with the name of that template.
- If you then open this activity, ClaimCenter 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, ClaimCenter 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 429 for details of working with the `activity-patterns.csv` file.

Localizing Activity Patterns

ClaimCenter 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 ClaimCenter interface—if you configure ClaimCenter for multiple locales:

- Description
- Subject

If you configure ClaimCenter 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 49 in the *Globalization Guide*.

Testing Gosu Code

Debugging and Testing Your Gosu Code

This topic discusses the code debugger that Guidewire provides in ClaimCenter Studio.

This topic includes:

- “The Studio Debugger” on page 435
- “Initializing the Application Server for Debugging” on page 436
- “Starting the Studio Debugger” on page 436
- “Setting Breakpoints” on page 436
- “Stepping Through Code” on page 437
- “Viewing Current Values” on page 437
- “Resuming Execution” on page 438
- “Using the Gosu Scratchpad” on page 438
- “Using the Gosu Scratchpad” on page 438
- “Suggestions for Testing Rules” on page 439

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 ClaimCenter 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 ClaimCenter 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.

Initializing the Application Server for Debugging

To use the Studio debugger functions, you must start the development server in debug mode. You do this with the following command:

```
gwcc -debug-start
```

Starting the Studio Debugger

You must start the debugger so if you want it to pause during execution of your code. If you do not start the debugger, then your code runs all the way through without pausing. You must also be connected to a running ClaimCenter server for the debugger to work. (If you do not have connection to a running ClaimCenter server, Studio prompt you to connect to one.)

To start the debugger, click **Debug Server** .

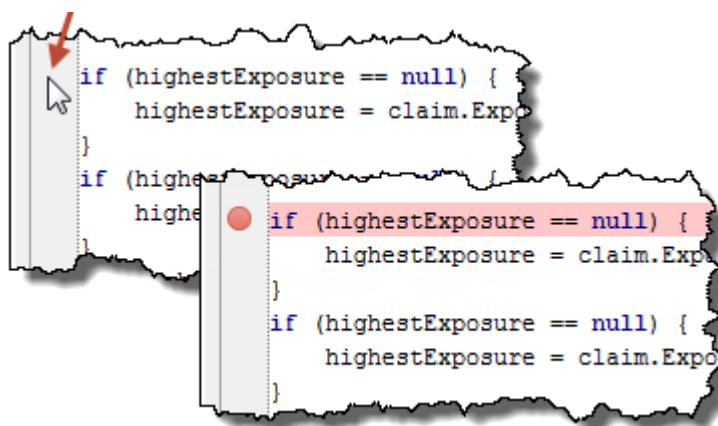
Studio shows the **Debug** pane.

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:

Step over	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 Step Over  .
Step into	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 Step Into  .

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 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.

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:

Stop debugging	Resume normal execution, and ignore all remaining breakpoints. To stop debugging, click Mute Breakpoints  , and then click Resume Execution  .
Continue debugging	Resume normal execution, but pause again at the next breakpoint, if any. To continue debugging, click Resume Execution  without having Mute Breakpoints  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 ClaimCenter 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 185 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 a code that contain 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 ClaimCenter 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 ClaimCenter 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 “Debugging and Testing Your Gosu Code” on page 435 for details.)

This topic includes:

- “The TestBase Class” on page 441
- “Configuring the Server Environment” on page 442
- “Configuring the Test Environment” on page 444
- “Creating a GUnit Test Class” on page 445
- “Using Entity Builders to Create Test Data” on page 448

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 448 for details on how to create test data.
- See “Creating a Builder for a Custom Entity and Testing It” on page 455 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.

ClaimCenter 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 441 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.
<code>@RunInDatabase</code>	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 <code>DatabaseForTest</code> values, specifying the databases which are specifically to be tested, or <code>DatabaseForTest.ALL</code> that allows the class to be run against any database.
<code>@RunLevel</code>	Allows a test to run at a different run level. The default value is <code>RunLevel.NO_DAEMONS</code> . 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">• <code>RunLevel.NONE</code> - Use if you do not want any dependencies at all.• <code>RunLevel.SHUTDOWN</code> - Use if you want all the basic dependencies set up, but with no database connection support.• <code>RunLevel.NO_DAEMONS</code> - Use for a normal server startup without background tasks. (This is also suitable for SOAP tests.)• <code>RunLevel.MULTIUSER</code> - Use to start a complete server (batch process, events, rules, Web requests, and all similar components).

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 444 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 JUnit .
	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 445 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 442 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 444.

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 ClaimCenter base configuration provides builders for the base entities (like ClaimBuilder, 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 ClaimCenter, 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()
```

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, PolicyPeriod is on an Account, 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 ClaimCenter 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.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 ClaimCenter 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 453
- “Extending the DataBuilder Class” on page 453
- “Creating a Builder for a Custom Entity and Testing It” on page 455

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 449):

Initial word	Indicates
on	A link to a parent, for example, PolicyPeriod is on an Account, 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>as AgencyBill</code> .
with	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 ClaimCenter 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 ClaimCenter configures that property on the target object. (ClaimCenter 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. ClaimCenter 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 implement 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 ClaimCenter 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 <code>ExtCredential</code> array entity and extend the <code>User</code> entity to include it.	<code>ExtCredential.eti</code> <code>User.etcx</code>	To create a custom entity
2. Create an <code>ExtCredentialBuilder</code> by extending the <code>DataBuilder</code> class and adding <code>withXXX</code> methods to it.	<code>ExtCredentialBuilder.gs</code>	To create an <code>ExtCredentialBuilder</code> class
3. Create a test class to exercise and test your new builder.	<code>ExtCredentialBuilderTest.gs</code>	To create an <code>ExtCredentialBuilderTest</code> 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 ExtCredential1.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 213 for information on extending the Guidewire ClaimCenter 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 453 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 204 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 `assertEquals` 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:

Guidewire ClaimCenter Configuration

Configuring Policy Behavior

There are several aspects of policy behavior that you can configure in ClaimCenter.

This topic includes:

- “Understanding Aggregate Limits” on page 461
- “Defining Aggregate Limits” on page 463
- “Advanced Aggregate Limit Configuration” on page 468
- “Specifying Policy Menu Links” on page 471
- “Defining Internal ClaimCenter Policy Fields” on page 471

Understanding Aggregate Limits

An aggregate limit is the maximum amount that an insurer is required to pay on a policy or coverage for a given period of time. An aggregate limit effectively caps the insurer’s total liability for the specified time period.

The management tasks associated with aggregate limits include:

- Defining calculation criteria, including cost types and cost categories, which count towards a limit
- Specifying the financial transactions that count towards a limit
- Configuring policy period definitions
- Manually recalculating how much of your aggregate limits have been used using ClaimCenter batch processes

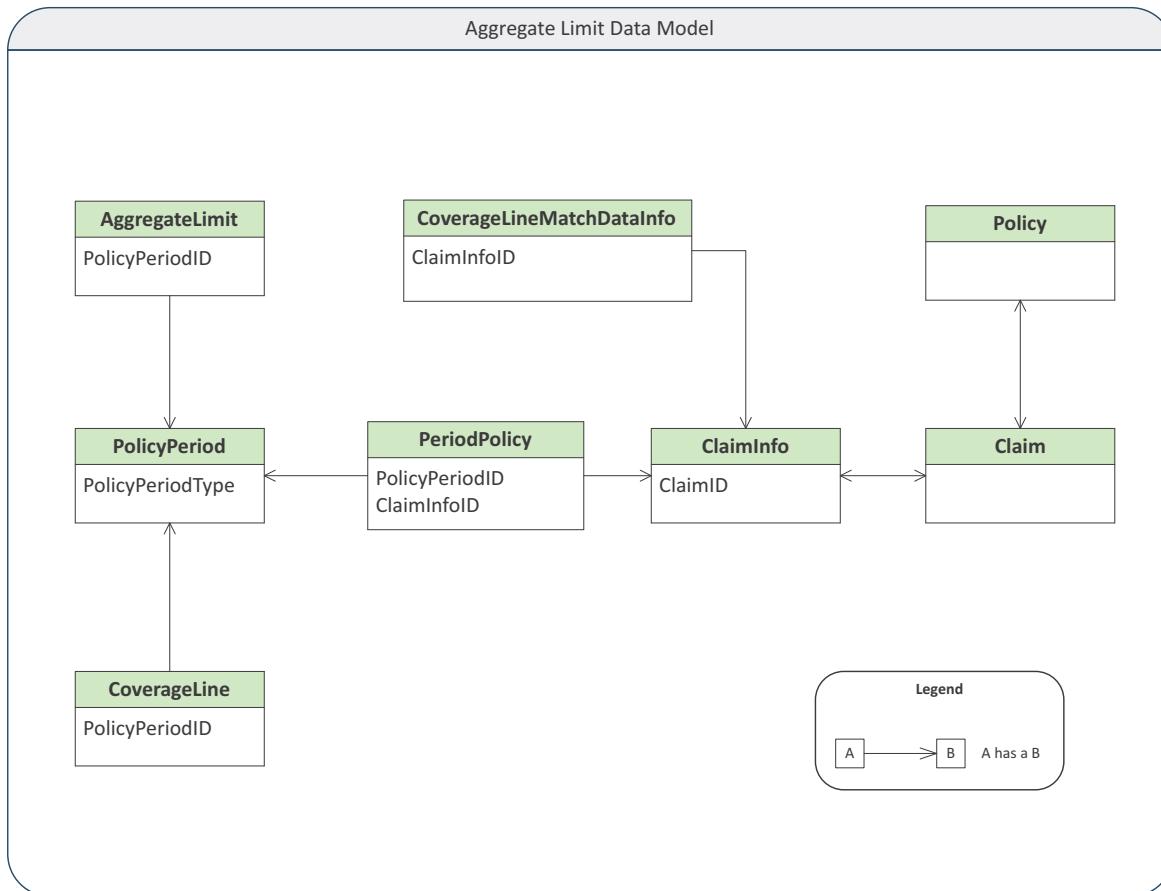
Note: If you want to use ClaimCenter to track and display aggregate limits, add the **Aggregate Limits** menu link to the relevant policy types. For more information, see “Specifying Policy Menu Links” on page 471.

At the highest level, an aggregate limit can apply to a policy or an account. A limit that applies to a single policy establishes a maximum total liability for all of the claims made against that policy. A limit that applies to an account establishes a maximum liability for all claims made against all the policies belonging to the account. The cap applies regardless of the number of claims made against the relevant policies or the number and variety of exposures represented in the claims.

For more about policy periods, see the section “Policy Periods” on page 103 in the *Application Guide*.

Aggregate Limit Data Model

The following graphic depicts a simplified view of how the `AggregateLimit` entity is related to other entities.



Claim Graph

`PolicyPeriod` and `AggregateLimit` and associated entities are cross-claim entities, which encompass multiple claims. They are not archivable and stay behind in the production database. Each `Claim` has one `Policy` object which holds policy data from the policy administration system and belongs to the claim. The `Policy` belongs to the claim graph and is archived with it. The `ClaimInfo` entity is not archived and is the connecting point between the admin/cross-claim data and the claim graph. Therefore, each `PeriodPolicy` (a join entity) points to its `PolicyPeriod` and to a `ClaimInfo` to indicate that the `Claim/Policy` is in the `PolicyPeriod`.

Policy Retrieval and Aggregate Limits

During the policy retrieval process, the `Claim` has not yet been associated with the `PolicyPeriod`. When the policy plugin returns policy data, the `Policy` is associated with the `Claim` and the `PolicyPeriod`. However, you do not have access to the `Claim` at this point.

As a result, you cannot add aggregate limits when using the policy search plugin. You can create limits later by either adding them to the claim from the user interface or using `ClaimPostsetup` rules.

For more information on the policy search plugin, see the section on “Policy Search Plugin” on page 517 in the *Integration Guide*.

Defining Aggregate Limits

You can define aggregate limits in the **Aggregate Limits** menu link of a claim's **Policy** screen.

The **AggregateLimit** entity has the following key properties:

Entity Property	Description
LimitAmount	The amount of the aggregate limit in the claim currency of claims in the policy period.
ValueType	Aggregate limit type – Limit or deductible.
AggLimitCalcCriteria	The calculation criteria, a cost type and cost category combination, to be specified for the aggregate limit. This property maps to the LimitUsedDef element definition in the aggregate limits configuration file, aggregatelimitused-config.xml .
PolicyPeriod	The policy period to which the aggregate limit belongs.
CoverageLines	Coverage lines that reference this aggregate limit. An aggregate limit can have multiple coverage lines or none at all.

Aggregate Limit Configuration

In Guidewire Studio, navigate to **configuration** → **config** → **aggregatelimit** to view and edit the following configuration files:

- **aggregatelimitused-config.xml** – Use this file to define what counts towards calculating an aggregate limit.
- **policyperiod-config.xml** – Use this file to define the policy period to associate with the aggregate limit.

These configuration options are described in more detail in the sections below.

Defining What Counts Towards an Aggregate Limit

ClaimCenter reports the **Realized** amount, the amount of an aggregate limit that has been eroded. In order to calculate this accurately, for each policy type that uses the aggregate limit, you must specify the following in the **aggregatelimitused-config.xml** file:

- The calculation type, **calctype**, defines the transaction types to count towards the limit.
- The calculation criteria, **AggLimitCalcCriteriaDefinition**, defines the possible options for cost types and cost categories to include or exclude in the limit.

The file consists of a main **AggregateLimitUsedConfig** element and one or more **AggLimitCalcCriteriaDefinition** and **LimitUsedDef** subelements.

Aggregate Limit Calculation Criteria Definition

The **AggLimitCalcCriteriaDefinition** element defines the combinations of cost types and cost categories that will be available as options when creating an aggregate limit. You can create custom calculation criteria to exclude different cost types and categories, based on your business requirements.

The **AggLimitCalcCriteriaDefinition** element has the following format:

```
<AggLimitCalcCriteriaDefinition code="calc_criteria">
    <ExclusionCriteria excludeCostType="cost_type" excludeCostCategory="cost_category"/>
</AggLimitCalcCriteriaDefinition>
```

Each **AggLimitCalcCriteriaDefinition** can have zero or more **ExclusionCriteria** subelements. This subelement, if included, must have a **excludeCostType** attribute specified. It may have an **excludeCostCategory** attribute. The **excludeCostType** and **excludeCostCategory** attributes enable you to specify cost types and cost categories for transactions to exclude in an aggregate limit calculation. The **costtype** and **costCategory** typelist define the possible types for **excludeCostType** and **excludeCostCategory**.

The code value must correspond to a typekey value in the AggLimitCalcCriteria typelist. You must define the calculation criteria in the typelist for it to appear in ClaimCenter when creating an aggregate limit.

The following conditions apply when defining aggregate limit calculation criteria:

- At least one AggLimitCalcCriteria definition is mandatory. In the base configuration, AggLimitCalcCriteria.tti contains all, which cannot be removed, but can be retired.
- You can have only one AggLimitCalcCriteriaDefinition entry marked as default. This covers cases where a policy type is not listed in any LimitUsedDef section.
- If AggLimitCalculationCriteriaDefinition uses a code that is not defined in the AggLimitCalcCriteria typelist, even if retired, the application server will fail to start and indicate the error in the configuration. This definition cannot be applied to any new aggregate limits, but existing ones will continue to work.

Note: The aggregate limits batch process needs access to the calculation criteria definition from the aggregatelimitused-config.xml file to recalculate the limit used, even if the typecode is retired.

- If an active code in the AggLimitCalcCriteria typelist is not defined by any AggLimitCalculationCriteria element, the server will fail to start.

Aggregate Limit Used Definition

The LimitUsedDef element defines the calculation type and default calculation criteria that count towards the limits of particular policies. You can create multiple LimitUsedDef elements to associate different policy types with different calculation type and calculation criteria. You must specify at least one.

The LimitUsedDef element definition has the following format:

```
<LimitUsedDef calctype="calc_type" aggLimitCalculationCriteriaDefault="calc_criteria">
  <AggLimitPolicyType code="policy_type"/>
</LimitUsedDef>
```

Every LimitUsedDef element contains one subelement, AggLimitPolicyType, and two attributes, calctype and aggLimitCalculationCriteriaDefault.

- The AggLimitPolicyType subelement specifies the policy types that share the same limit definition. You must list at least one policy for a definition. The code attribute defines the policy type. The PolicyType typelist defines the list of possible types. A policy type can only be included in one LimitUsedDef element.
- The calctype attribute defines the financial calculation to apply while determining the realized amount of an aggregate limit. The calctype can be one of the following:

Type	Description
TotalIncurredGross	The sum of total reserves plus non-eroding payments.
TotalIncurredNet	The sum of total reserves plus non-eroding payments minus recoveries.
TotalPayments	The sum of all submitted payments and awaiting-submission payments whose scheduled send date is either before or on the current date.
TotalIncurredNetMinusOpenRecoveryReserves	The sum of total reserves excluding recovery reserves that are still open.
TotalPaymentsPlusFuturePayments	TotalPayments plus payments on future-dated checks.
TotalIncurredNetPlusFutureNonErodingPayments	TotalIncurredNet plus non-eroding payments on future-dated checks.
TotalIncurredNetMinusOpenRecoveryReservesPlusFutureNonErodingPayments	TotalIncurredNetMinusOpenRecoveryReserves plus non-eroding payments on future-dated checks.
TotalIncurredGrossPlusFutureNonErodingPayments	TotalIncurredGross plus non-eroding payments on future-dated checks.

- The `aggLimitCalculationCriteriaDefault` attribute defines the default calculation criteria to be used towards this limit. The value of this attribute must match the code of one of the previously defined `AggLimitCalcCriteriaDefinition` subelements.
- The `AggLimitCalcTypeDefault` element defines the default calculation type to use if a policy type is not listed in the `LimitUsedDef` element. The calculation type determines the transaction subtypes to apply towards the limit used in the aggregate limit. In the case of account-level aggregate limits, the calculation type is the system default.

Aggregate Limit Configuration: Example

In the following example, the `LimitUsedDef` element defines the costs for commercial and personal automobile policies. Three custom calculation criteria are defined, `all` (default), `costTypesExcludingLegalExpenses`, and `costTypesExcludingExpenses`.

```
<AggregateLimitUsedConfig
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="aggregatelimitused-config.xsd">

    <AggLimitCalcCriteriaDefinition code="all" default="true"/>

    <AggLimitCalcCriteriaDefinition code="costTypesExcludingExpenses">
        <ExclusionCriteria excludeCostType="unspecified"/>
        <ExclusionCriteria excludeCostType="dcexpense"/>
        <ExclusionCriteria excludeCostType="aoexpense"/>
    </AggLimitCalcCriteriaDefinition>

    <AggLimitCalcCriteriaDefinition code="costTypesExcludingLegalExpenses">
        <ExclusionCriteria excludeCostType="unspecified" excludeCostCategory="legal"/>
        <ExclusionCriteria excludeCostType="aoexpense" excludeCostCategory="legal"/>
        <ExclusionCriteria excludeCostType="dcexpense" excludeCostCategory="legal"/>
    </AggLimitCalcCriteriaDefinition>

    <AggLimitCalcTypeDefault defaultCalctype="TotalIncurredNet"/>

    <LimitUsedDef
        calctype="TotalIncurredNet"
        aggLimitCalculationCriteriaDefault="costTypesExcludingExpenses">
        <AggLimitPolicyType code="PersonalAuto"/>
        <AggLimitPolicyType code="BusinessAuto"/>
    </LimitUsedDef>

</AggregateLimitUsedConfig>
```

For the policy types included in the `LimitUsedDef` subelement, the total incurred net value of financial transactions is used in the aggregate limit calculation. The calculation criteria for aggregate limits defined for these policy types will default to the corresponding value in the typelist for `costTypesExcludingExpenses`. You can still choose to change the calculation criteria while creating the aggregate limit.

From this, ClaimCenter calculates an aggregate limit's used amount as follows:

- For each transaction that belongs to the policy period attributed to the limit, ClaimCenter determines its claim's policy type.
- If the configuration file *does not* list the policy type, then the transaction does not contribute to the limit used calculation.
- If the configuration file *does* list the policy type, ClaimCenter determines whether the `LimitUsedDef` to which the policy type belongs lists the transaction's cost type in `aggLimitCalculationCriteriaDefault`.
 - If `LimitUsedDef` *does not* list the cost type, then it does not contribute to the limit used calculation.
 - If `LimitUsedDef` *does* list the cost type, then ClaimCenter retrieves the calculation type associated with cost type.

Then, ClaimCenter determines whether that the transaction applies to the aggregate limit (and how to calculate the transaction). It then calculates the amount and applies it to the limit used for that aggregate limit.

IMPORTANT Changing the `aggregatelimitused-config.xml` file can have multiple implications. You might have to run the aggregate limits batch process again, depending on the type of change.

See Also

- “Aggregate Limit Calculations” on page 130 in the *System Administration Guide*

Configuring Policy Periods

A policy period identifies the policy or policies that count towards an aggregate limit. The policy period does this by defining the following:

- The effective time period for the aggregate limit
- The set of policy data fields that must be identical for the policy to count towards an aggregate limit

In ClaimCenter, policy periods are created as needed for aggregate limits, based on the definitions in the `policyperiod-config.xml` file. The definitions in this file function as the blueprint for your policy periods. A *policy period definition*, `PolicyPeriodDef`, is a configuration object that determines the form and functionality of one or more policy periods.

ClaimCenter bases aggregate limits on policy periods. A specific and unique policy period identifies the claims that count towards an aggregate limit. When a policy is created, ClaimCenter searches for an existing policy period that matches the properties of the policy. If there is no existing policy period that matches the policy, ClaimCenter uses the appropriate policy period definition as a template to create the necessary policy period.

WARNING After you create a policy period definition and ClaimCenter begins to use it, do **not** attempt to change it. Guidewire does **not** support updating policy periods to reflect a new `policyperiod-config.xml` definition.

The syntax of a policy period definition is as follows:

```
<PolicyPeriodDef type="type">
  <PolicyTypeConfig code="policy_type"/>
  ...
  <PolicyField fieldName="fieldName"/>
  ...
</PolicyPeriodDef>
```

This definition uses the following elements:

- The `PolicyPeriodDef` element defines an individual policy period. Within this element, you must specify at least one `PolicyTypeConfig` element and one `PolicyField` element.

Note: If you do not define at least one `PolicyPeriodDef`, you effectively disable aggregate limits in your installation.

- The `type` parameter takes one of two values: `policy` or `account`.
- The `PolicyTypeConfig` element specifies the policy type to which the definition applies. This is a code from the `PolicyType` typelist. `PolicyPeriodDef` must include a `PolicyTypeConfig` element for each policy type that you want to include in the definition.
- The `PolicyField` element specifies a policy data field to include in the period definition. There are five allowable fields, all of which are also fields of the `PolicyPeriod` entity.
 - `AccountNumber` – Used with policy period definitions of type “account.” A case-sensitive text field in a policy’s definition identifies the name or number of the account to which the policy belongs.
 - `PolicyNumber` – The alphanumeric value that ClaimCenter uses to identify a policy.
 - `EffectiveDate` – The date on which an individual policy goes into effect.
 - `ExpirationDate` – The date on which an individual policy goes out of effect.
 - `PolicySuffix` – The unique, alphanumeric value that you append to a policy number to differentiate between effective years for a policy, also called `Mod` or `Module`. Typically, you use this only for policy periods of type `policy`.

The `PolicySuffix` field acts as an implicit time period. It identifies both an effective date and an expiration date. By including the suffix, you ensure that claims made in different years are not mistakenly aggregated into the same limit.

Policy-based Period Definition: Example

The following `PolicyPeriodDef` example ensures that all claims made against a specific automobile policy count against aggregate limits defined for that policy period or policy term.

```
<PolicyPeriodDef type="policy">
  <PolicyTypeConfig code="PersonalAuto"/>
  <PolicyTypeConfig code="BusinessAuto"/>

  <PolicyField fieldName="PolicyNumber"/>
  <PolicyField fieldName="PolicySuffix"/>
</PolicyPeriodDef>
```

This definition specifies that a single aggregate limit applies for all policies that meet the following criteria:

- All are either of type personal auto or commercial auto *and*
- All have the same policy number *and*
- All have the same policy suffix

Example of Account-based Period Definition

The following `PolicyPeriodDef` example ensures all policies belonging to the same account count towards aggregate limits.

```
<PolicyPeriodDef type="account">
  <PolicyTypeConfig code="GeneralLiability"/>
  <PolicyTypeConfig code="CommercialProperty"/>
  <PolicyTypeConfig code="InlandMarine"/>
  <PolicyTypeConfig code="Homeowners"/>

  <PolicyField fieldName="AccountNumber"/>
  <PolicyField fieldName="EffectiveDate"/>
  <PolicyField fieldName="ExpirationDate"/>
</PolicyPeriodDef>
```

This definition specifies *all* policies included in the limit meet the following criteria:

- All are of type general liability, commercial property, inland marine, or homeowners LOB *and*
- All have the same `AccountNumber` value *and*
- All have the same effective date *and*
- All have the same expiration date

Storing Aggregate Limit Data

ClaimCenter uses the entity, `ClaimAggregateLimitRpt`, to store the amount used for each claim that contributes to an aggregate limit. If the aggregate limit uses a coverage line, it is stored in `ClaimAggregateLimitRpt` along with the corresponding contribution amount.

The `ClaimAggregateLimitRpt` entity is associated with a `PolicyPeriod`, `ClaimInfo`, and optionally, a `CoverageLine` object. Consequently, the limit used amounts stored in this entity are updated dynamically based on ClaimCenter changes. Some examples are when a claim moves from one policy period to another, when the loss date or reported date changes, or when transaction information is updated.

This entity is not in the claim graph, and this enables ClaimCenter to use and edit aggregate limit data even when one or more of the contributing claims are archived.

Aggregate Limit Used Recalculation

Any time you make relevant changes to claim data, ClaimCenter automatically recalculates how much of an aggregate limit is used. As a general rule, you do not need to manually recalculate the realized portion of your limits. However, if it becomes necessary for you to perform a manual recalculation, you can do so.

If any of the database consistency checks related to aggregate limits fail, you need to instruct ClaimCenter to recalculate the aggregate limit values. You also need to instruct ClaimCenter to repopulate the database tables. The following commands are used to recalculate aggregate limits:

- To rebuild aggregate limits marked as invalid in the database:

```
maintenance_tools -rebuildagglimits
```

This command finds all policy periods with at least one invalid aggregate limit and rebuilds all limits within that period.

- To rebuild aggregate limits for a single claim or one or more comma-separated claims:

```
maintenance_tools -rebuildagglimits -claims <claimnumbers>
```

- To rebuild aggregate limits for a single policy or one or more comma-separated policies:

```
maintenance_tools -rebuildagglimits -policies <policynumbers>
```

- To rebuild all aggregate limits in the system:

```
maintenance_tools -rebuildagglimits -forceall
```

Note: The system must be in maintenance mode for the `-forceall` parameter.

For information on the aggregate limits batch processes, see “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide*.

Advanced Aggregate Limit Configuration

Aggregate limits can be configured using the definitions in the configuration file, `aggregateLimitUsed-config.xml`, in conjunction with any user-defined selections of coverage types, subtypes, and covered items in the user interface. In addition, ClaimCenter also allows you to extend the configuration file using the `AggregateLimitTransactionPlugin`. It provides additional flexibility for those who wish to include custom Gosu code to specify transaction or claim eligibility for an aggregate limit.

This topic describes the `AggregateLimitTransactionPlugin` and illustrates its use with some configuration examples.

Using the `AggregateLimitTransactionPlugin`

The `AggregateLimitTransactionPlugin` determines the eligibility of a transaction to an aggregate limit. You can create your own custom plugin implementation, which delegates to the base implementation, as needed.

IMPORTANT Deploying a new implementation of the `AggregateLimitTransactionPlugin` might potentially impact many aggregate limits. After any such changes, you are responsible for updating all limits. Even a forced rebuilding of all limits cannot accurately recalculate values for aggregate limits that reference archived claims.

The `AggregateLimitTransactionPlugin` plugin has two responsibilities:

- Given a transaction and an aggregate limit, determine whether the transaction applies toward the limit.
- Determine if a change to a transaction or related entity might alter the applicability of a transaction to some limit.

When transaction-related information changes, ClaimCenter needs to know if any aggregate limits must be recalculated. ClaimCenter has built-in checks for known changes that trigger recalculation, such as changing the

amount of a transaction, or adding a new coverage line to an aggregate limit. However, if custom logic has been added to determine applicability, then a completely different type of change may need to trigger recalculation.

This plugin enables customers to communicate the entity types that can possibly trigger a change in applicability and whether a change to one of those entities impacts applicability.

The plugin uses the standard aggregate limit method, `configurationAppliesTo(transaction)`, which returns a boolean value, to evaluate transactions. In the base configuration, this method simply refers to `aggregatelimitused-config.xml` and determines whether the transaction applies to the aggregate limit. You can include additional, custom logic to extend the configuration defined in `aggregatelimitused-config.xml` by using the methods detailed next.

To determine if a change to the transaction or a related entity impacts the applicability of the transaction to the aggregate limit, the following two methods are used:

- `claimWithChangedTransactionApplicability()` – This method examines the given modified entity to determine if the change impacts the eligibility of associated transactions to a specific aggregate limit. If the modification impacts transactions contributing towards the aggregate limit calculation, the method returns the corresponding claim. ClaimCenter then recalculates the aggregate limit.

This check is for non-standard changes that might impact aggregate limit calculation. ClaimCenter already checks a predefined set of fields, such as a transaction's amount, that might trigger recalculation. The purpose of this method is to define checks for customizations and extensions to the base configuration.

This method will only be called for a modified entity whose type is included in `typesWhichAffectApplicability()`.

- `typesWhichAffectApplicability()` – This method specifies the modified entity types that are called by `claimWithChangedTransactionApplicability()`. In the base configuration, this method returns a set of entities of the type Transaction.

See also

- “Aggregate Limit Used Recalculation” on page 468.

Example 1. Configuring Claims for Aggregate Limits

In this example, the `AggregateLimitTransactionPlugin` is used to extend the base configuration to exclude or include a claim from being applied towards aggregate limits. It preserves the configuration checks included in `aggregatelimitused-config.xml` and adds an additional check for a custom, claim-level flag that turns aggregate limit eligibility on or off for the claim.

Use the following steps to implement this example.

1. If necessary, start Guidewire Studio.

At a command prompt, navigate to the `ClaimCenter/bin` directory and enter:

```
gwcc studio
```

2. Navigate to `configuration → config → Extensions → Entity` and open `Claim.etcx`.

3. Click the plus icon (), and select `column` from the drop-down choice list.

4. Enter the following values:

Name	Value
name	AggregateLimitsApply
type	bit
desc	Flag that indicates if this claim applies towards aggregate limits.

5. Compile and save your changes.

6. Add a custom Gosu plugin implementation, as follows:

```

/*
 * Implementation that uses a custom AggregateLimitsApply field to turn aggregate limits
 * on/off per claim.
 */
class ExampleAggregateLimitPlugin implements IAggregateLimitTransactionPlugin
{
    var _delegate = new AggregateLimitTransactionPluginImpl()

    override function aggregateLimitApplies(limit: AggregateLimit, transaction: Transaction) : boolean
    {
        // Verifies the transaction against the configuration and also checks the AggregateLimitsApply
        // field.
        return _delegate.aggregateLimitApplies(limit, transaction)
            and transaction.Claim.AggregateLimitsApply
    }

    override function typesWhichAffectApplicability() : Set<IEntityType>
    {
        // Adds Claim to the set of types which can affect applicability.
        return _delegate.typesWhichAffectApplicability().union({Claim})
    }

    override function claimWithChangedTransactionApplicability(modifiedEntity: KeyableBean) : Claim
    {
        // Check base configuration and then, also check if the AggregateLimitsApply field has changed.
        var affectedClaim = _delegate.claimWithChangedTransactionApplicability(modifiedEntity)
        if (affectedClaim == null)
        {
            if (modifiedEntity typeis Claim and modifiedEntity.isFieldChanged(Claim#AggregateLimitsApply))
            {
                affectedClaim = modifiedEntity
            }
        }
        return affectedClaim
    }
}

```

7. Save your changes.

Example 2. Configuring Aggregate Limits for Deductible Recoveries

In this second example, an `AggregateLimitTransactionPlugin` is used to configure aggregate limit calculations to only include deductible recoveries. If the limit type is deductible, only recoveries with the category, deductible, contribute to the aggregate limit. For all other aggregate limits, whose limit type is `limit`, the base configuration implementation will be in effect.

Use the following steps to implement this example.

1. If necessary, start Guidewire Studio.

At a command prompt, navigate to the `ClaimCenter/bin` directory and enter:

```
gwcc studio
```

2. Override the plugin implementation, as follows:

```

class ExampleAggregateDeductiblePlugin implements IAggregateLimitTransactionPlugin
{
    static final var WITH_LIMITS = RecoveryCategory.TC_DEDUCTIBLE
    static final var deductibleType = AggregateType.TC_DEDUCTIBLE
    var _delegate = new AggregateLimitTransactionPluginImpl()

    override function aggregateLimitApplies(limit: AggregateLimit, transaction: Transaction) : boolean
    {
        if (limit.ValueType != deductibleType)
            return _delegate.aggregateLimitApplies(limit, transaction)
        else
            return _delegate.aggregateLimitApplies(limit, transaction)
                and transaction.RecoveryCategory == WITH_LIMITS
    }

    override function typesWhichAffectApplicability() : Set<IEntityType>

```

```
{  
    return _delegate.typesWhichAffectApplicability().union({Recovery})  
}  
  
override function claimWithChangedTransactionApplicability(modifiedEntity: KeyableBean) :Claim  
{  
    var affectedClaim = _delegate.claimWithChangedTransactionApplicability(modifiedEntity)  
    if (affectedClaim == null)  
    {  
        if (modifiedEntity typeis Recovery and recoveryCategoryChangedToOrFromNoLimits  
            (modifiedEntity))  
        {  
            affectedClaim = modifiedEntity.Claim  
        }  
    }  
    return affectedClaim  
}  
  
private function recoveryCategoryChangedToOrFromNoLimits(recovery: Recovery): boolean  
{  
    return recovery.RecoveryCategory== WITH_LIMITS  
        or recovery.getOriginalValue(entity.Recovery#RecoveryCategory) == WITH_LIMITS  
}
```

3. Save your changes.

Specifying Policy Menu Links

You can customize the menu links that appear on the **Policy** menu of a claim file. For example, an automobile policy can include a menu link with a list of vehicles and a property policy can include a link with a list of locations.

The **Policy** page always contains the **General** menu link, and you can add one or more other options. The **PolicyTab** typelist defines available menu links that you can add.

You can specify the menu links in the **PolicyType** typelist in Guidewire Studio. To define a policy type, use the **category** tab to add a typelist and typecode from the **PolicyTab** typelist.

In addition, if the menu link you want to add is **Aggregate Limits**, you must add the policy type to the following two configuration files:

- **aggregateLimitUsed-config.xml** – For more information on this configuration file, see “Aggregate Limit Configuration” on page 463.
- **policyperiod-config.xml** – For more information on this configuration file, see “Configuring Policy Periods” on page 466.

Defining Internal ClaimCenter Policy Fields

Typically, applications external to ClaimCenter store the primary policy data. Thus, ClaimCenter imports policy snapshots for use with claims. Depending on the status of the policy within ClaimCenter, it can be classified as verified or unverified.

- A *verified* policy in ClaimCenter reflects the policy from the external application. You cannot modify it in ClaimCenter.
- An *unverified* policy does not necessarily reflect the external application.

You can use the status of a policy to in your configuration of the application. For example, you might configure validation rules so that you can only make payments on a claim that has verified policies associated with it.

In reality, it is often useful to have a hybrid policy structure. With a hybrid structure, you can maintain a connection to an external application with a verified policy and add additional policy data fields for use in ClaimCenter. You add these data fields as *internal-only* fields to a policy, and ClaimCenter manages these internally. It does

not change or delete them if you refresh the policy snapshot from the external application or even if you change to a different policy. You can also edit internal fields without causing the policy to become unverified.

Creating Internal-Only Fields

While the base configuration does not contain internal-only fields, you can define them in the **Policy** entity file extension (**Policy.etcx**) and modify the targeted PCF files in Guidewire Studio. The **ClaimPolicyGeneral.pcf** file contains multiple **Edit** buttons, each one providing a different level of control over editing policies.

You can edit:

- An unverified policy
- A verified policy without internal-only fields
- Internal-only fields on the policy if the user does not have permission to edit a verified policy
- Optionally edit the entire policy or its internal-only fields if the policy has such fields and the user has the permissions to edit them

This example explains how to add a custom column on an auto policy.

1. Right-click the **Policy.eti** file and select **New → Entity Extension** to create **Policy.etcx**.
2. Internal-only fields can be regular columns, foreign keys, or arrays. Use the **<internalonlyfields>** element to list the fields that are internal to ClaimCenter, and use the **<internalfield>** element to identify a particular field. You can list multiple **<internalfield>** elements, with each specifying one field.

Define your internal-only field in that file. Your code might look like the following:

```
<extension xmlns="http://guidewire.com/datamodel" entityName="Policy">
  <column name="CustomColumn" type="varchar">
    <columnParam name="size" value="30"/>
  </column>
  <internalonlyfields>
    <internalfield name="CustomColumn" />
  </internalonlyfields>
</extension>
```

3. Save your work.
4. Open the **PolicyGeneralPanelSet_Auto.pcf** file and add a new field that is internal-only. In the **Basic properties** tab, ensure that **editable** is set to **true** and assign a unique **label** and **value**. This example uses the label **Internal Notes** with a value of **Policy.Notes** and the **id** is **CustomColumn**.
5. Since **required** is set to **true** in **Basic properties**, and this is not necessary, remove it.
6. Save your work in Studio and restart the application server.

Verify your change

Test your configuration using the steps below:

1. Open an auto claim in ClaimCenter and navigate to the **Policy** menu on the left pane. Select it to view the **Policy:General** screen.
2. If you select **Edit**, you can either modify the entire policy or ClaimCenter-only fields. Select the latter.
3. Edit the field you created and save your changes.

In the **Policy:General** screen, notice that under the **Other** section, the **Verified Policy** field is still set to **Yes**.

Configuring Snapshot Views

This topic explains how to configure snapshot views for use with your claims.

This topic includes:

- “How ClaimCenter Renders Claim Snapshots” on page 473
- “Encrypting Claim Snapshot Fields” on page 474
- “Configuring Snapshot Templates” on page 475

How ClaimCenter Renders Claim Snapshots

A first notice of loss (FNOL) snapshot is a persistent copy of a claim and the graph of entities it references. A FNOL snapshot records the data available to a carrier at the time an insured first notifies the carrier of a loss. ClaimCenter creates a FNOL snapshot as you create a claim using the **New Claim** wizard or as you import a claim into ClaimCenter from an external FNOL application.

Note: ClaimCenter takes FNOL snapshots by default. You can prevent ClaimCenter from creating FNOL snapshots by setting the `config.xml` parameter `EnableClaimSnapshot` to `false`.

The purpose of a FNOL snapshot is to retain the claim data in its original form regardless of any subsequent changes to the claim record. The data remains static to reflect the FNOL precisely at the moment of creation. You can view the FNOL snapshot for a claim by clicking the **FNOL Snapshot** page action in the **Loss Details** page for that claim.

Because the information contained in a FNOL snapshot does not change, ClaimCenter stores a snapshot in the `ClaimSnapshot` entity. The application creates a XML representation of the actual data and places the data in the `ClaimData` field of this entity. The entity also tracks the ClaimCenter version used to capture it.

If ClaimCenter renders a FNOL snapshot, it renders the snapshot data in a format similar to the **Loss Details** page. It is possible that between versions of ClaimCenter, the default fields that accompany claims change. For this reason, the `ClaimData` object can capture different data between releases. To support this, ClaimCenter contains version-specific PCF files to render the FNOL snapshot. You can access these files, through Studio, in the following location:

Resources → Page Configuration (PCF) → claim → snapshot

Understanding Snapshot PCF Interaction

At the top of the PCF → claim → snapshot folder are a number of modal PCF files. These files are modal because, based on the original version of the snapshot, ClaimCenter retrieves a version-specific PCF file from the underlying subdirectories. For example, if you select FNOL Snapshot in ClaimCenter, ClaimCenter opens the modal ClaimSnapshotLossDetails file.

This PCF declares a variable Snapshot whose initial value is the snapshot corresponding to the Claim variable. At this point, ClaimCenter cannot determine the object type of the snapshot. It is important to understand that the snapshot object type is unknown to ClaimCenter within the top-level claim snapshot pages. Thus, you must cast the snapshot to the correct type before you pass it into any shared list or detail view.

ClaimCenter uses a method to retrieve the Version value from the snapshot data. The mode attribute of the ScreenRef element uses the Version to locate the appropriate PCF file. If you review the Studio subdirectories, you see the following possible matches for this modal call:

- ClaimSnapshotLossDetailsScreen.300.pcf
- ClaimSnapshotLossDetailsScreen.310.pcf
- ClaimSnapshotLossDetailsScreen.400.pcf
- ...

For example, if you request snapshots created with ClaimCenter 3.0, the application renders the loss details using the 300 → ClaimSnapshotLossDetailsScreen.300 file.

This file takes as arguments the Claim and the snapshot (SnapshotParam) in turn. The code declares the SnapshotParam type as snapshot.v300.Claim

Note: Beginning with ClaimCenter version 3.0, Guidewire began supporting the rendering of FNOL snapshots in version-specific pages. Guidewire provides PCF files for each major version of ClaimCenter since 3.0 and plans to continue to do so in future release. Thus, if the next release of ClaimCenter happens to be 6.0, the release includes all the templates for each major released version between 3.0 and 6.0.

Encrypting Claim Snapshot Fields

ClaimCenter provides field level encryption on sensitive data, such as tax IDs. Those fields are also encrypted on claim snapshots.

There are two configurable components to this process:

- *Your algorithm called by the IEncryption plugin.* In this case, you must provide an algorithm.
- *The Encryption Upgrade work queue.* You can configure this work queue to run at a different time. You can also determine the number of snapshots that are upgraded at one time.

If you use claim snapshots and you decide to change your encryption algorithm, the Encryption Upgrade work queue updates those encrypted fields using the most current encryption algorithm.

See also

- “List of Batch Processes and Distributable Work Queues” on page 129 in the *System Administration Guide* to understand how the Encryption Upgrade work queue functions.
- “Encryption Integration” on page 249 in the *Integration Guide* to learn how to encrypt your data using the IEncryption plugin.

Configuring Snapshot Templates

ClaimCenter bases the FNOL snapshot pages that it provides on the ClaimCenter default elements. For example, the `ClaimSnapshotLossDetailsScreen.500.pcf` page renders the same basic **Loss Details** page as the `PCF → claim → ClaimLossDetails` page.

If you extend the basic loss detail pages for any release, you also need to update any corresponding claim snapshot pages. In a similar manner, if you add new exposure fields in a release, then you also need to update the corresponding exposure snapshot pages.

Configuring snapshot pages can involve several possible tasks:

- Editing snapshot PCF files to change the values that appear in a rendered page.

This is the most common FNOL snapshot configuration task. Typically, you want data rendered in a snapshot page to be the same as—and to be presented in the same way—as data in the corresponding data view. This involves adding or removing entity listings to, or from, pages as needed.

- Adding a new page or panel view.

This situation usually arises if you have added one or more custom pages. If you add a custom page, you also need to modify the appropriate pages to reference your new page.

IMPORTANT Guidewire strongly recommends that you use source control or a backup to save original versions of the ClaimCenter default snapshot templates before you modify them.

Snapshots and Data Model Extensions

Snapshot PCF files do not access the ClaimCenter database. Instead, ClaimCenter extracts the snapshot data from a text column on the `ClaimSnapshot` object. This field contains an XML description of the claim's FNOL. This has implications for data model extensions and changes.

If you extend or change the ClaimCenter data model *before* the application captures a FNOL snapshot, ClaimCenter captures your extension in the snapshot. That is, if there is actually data to capture. However, pre-existing snapshot data does *not* capture any data model extensions or changes that you make.

Note: If a particular snapshot does not contain the data your page references, then ClaimCenter displays the FNOL snapshot fields as empty.

Configuring Lines of Business

A line of business (LOB) is a business unit that is independent of other business units in a company. ClaimCenter enables you to model the responsibilities of each business unit of an insurance carrier. When you model your business structure, you also configure ClaimCenter screens and methods of handling claims for each part of your business. You model your business structure by configuring the relationships between a set of typelists:

- **LossType** and **LOBCode** – Model which business units cover each kind of loss.
- **LOBCode** and **PolicyType** – Model the kinds of policies written by each business unit.
- **PolicyType** and **CoverageType** – Model the types of coverages that policies can contain.
- **CoverageType** and **CoverageSubtype** and **ExposureType** – Model the kinds of exposures that are compatible with coverages.

In the base configuration, the LOB typelists contain values for the following lines of business:

- Businessowners Line
- Commercial Auto Line
- Commercial Property Line, which includes Commercial Package and Commercial Property policy types
- General Liability Line
- Homeowners Line
- Inland Marine Line
- Other Liability Line, which includes Farmowners and Professional Liability policy types
- Personal Auto Line
- Personal Umbrella Line
- Travel
- Workers' Comp Line

You can define new lines of business by adding to **LOBCode** and the other LOB typelists.

This topic includes:

- “LOB Typelists” on page 478
- “Relationships Among LOB Typelists” on page 479

- “Relationships Between LOB Typelists and Other Typelists” on page 482
- “Editing LOB Typelists and Typecodes” on page 484
- “Coverages and Policies” on page 489
- “Adding a New LossType Typecode” on page 491
- “Adding a New ExposureType Typecode” on page 492

LOB Typelists

ClaimCenter uses a set of typelists, the *line of business* typelists, to configure the screens you see when entering a new claim and working with existing ones. You can see these typelists in ClaimCenter Studio by navigating in the Project window to **configuration** → **config** → **Extensions** → **Typelist**.

Because the six typelists have a hierarchical relationship, the **LOB** tab of the Typelists editor displays the values of the typelists as a tree. For example, with the **LossType** typelist open in the editor, the **LOB** tab displays a number of **LOBCode** children for each **LossType** typecode. Each **LOBCode** child has **PolicyType** children, and so on.

In the base configuration, ClaimCenter provides the following LOB typelists, listed in hierarchical order:

- **LossType.ttx** – A category of activity that is an industry standard for insurance activity, such as Auto, Liability, Property, or Workers’ Compensation. Selecting a loss type is one of the early steps in defining a claim. It is a way of grouping your lines of business together.
A **LossType** typecode does not have a parent, but it can have multiple children that are **LOBCode** typecodes. The relationship of **LossType** to **LOBCode** is one to many.
- **LOBCode.ttx** – A type of work handled by one business unit of an insurance company. In many cases, a claim’s **LOBCode** is uniquely determined by the combination of its **LossType** and **PolicyType**. If it is not, the **LOBCode** can be selected in the **Loss Details** screen of the New Claim wizard and the claim. More than one LOB can handle the same type of loss. For example, both personal and commercial auto LOBs sell policies covering losses from vehicles, the Auto loss type.

An **LOBCode** typecode can have multiple children from the **PolicyType** typelist, but only one parent, a **LossType** typecode. The relationship of **LOBCode** to **LossType** is many-to-one.

- **PolicyType.ttx** – A product—policy—sold by an LOB unit. A policy pays damages for certain defined loss events. One business unit can sell many types of policies. A given policy can be sold by many LOBs, so **PolicyType** and **LOBCode** have a many-to-many relationship in the data model.
- **CoverageType.ttx** – A subdivision of a policy that deals with one kind of loss covered by the policy. For example, a personal injury protection (PIP) policy could have medical, lost wages, and death coverages, so you can associate many coverages with a specific policy. These coverages define the product that is sold. Also, a particular type of coverage can be allowed for many policies. Thus, **PolicyType** and **CoverageType** are related in a many-to-many relationship.
- **CoverageSubtype.ttx** – Specifies an **ExposureType** that is compatible with a **CoverageType**. An **ExposureType** is a set of information collected in the ClaimCenter user interface when creating a exposure. A **CoverageSubtype** typecode acts like a join between **CoverageType** and **ExposureType**. When the ClaimCenter user selects a **CoverageSubtype** as part of creating an exposure, they are really setting the **ExposureType**. The **CoverageSubtype** the user selects becomes a property of the **Exposure** and not the **Coverage**.

This typelist effectively relates each **CoverageType** to one or more **ExposureType** typecodes. **CoverageSubtype** has many-to-one relationships to both **CoverageType** and **ExposureType**, so it effectively creates a many-to-many relationship between **CoverageType** and **ExposureType**. By convention, if a **CoverageType** is compatible only with one **ExposureType**, then the **CoverageSubtype** linking them is named after the **CoverageType**. Otherwise, its name is a concatenation of the **CoverageType** and a string indicating the **ExposureType**.

- `ExposureType.ttx` – `ExposureType` controls the mode of the screen used to create, display, or edit an exposure. This use of modes enables a relatively small number of user interface elements to be shared by exposures with a much larger number of possible coverage subtypes. In a claim, exposures relate coverages to claimants, and an exposure is a unique combination of a coverage and a claimant. For example, an auto accident claim would typically have an exposure for the owner of each vehicle damaged, as well as one for each person injured. An `ExposureType` cannot have any children. It has a one-to-many relationship with its parent typelist, `CoverageSubtype`.

See also

- “Relationships Among LOB Typelists” on page 479
- “Relationships Between LOB Typelists and Other Typelists” on page 482
- “Editing LOB Typelists and Typecodes” on page 484
- “Coverages and Policies” on page 489
- “Adding a New LossType Typecode” on page 491
- “Adding a New ExposureType Typecode” on page 492

LOB Typelists and Policies

Information about the `LossType`, `LOBCode`, `PolicyType`, `CoverageType`, and `CoverageSubtype` is ultimately determined by the policy administration system, such as Guidewire PolicyCenter. ClaimCenter maintains local information about the structure of a policy, but this information maps to equivalent information in the policy administration system. When ClaimCenter retrieves a policy from the policy administration system, ClaimCenter uses its local information to interpret and store the policy information.

Information about `ExposureType` is unique to ClaimCenter. `ExposureType` is a mechanism used to streamline the gathering of information about a loss. Therefore, there is no analog to exposure type in the policy administration system.

LOB Typelists and Incidents

While not technically part of the LOB model, the `Incident` typelist, which represents subclasses of the `Incident` entity, also relates to the LOB typelists. Every non-final entity type automatically has a corresponding typelist with the same name whose values represent each of the subclasses of the entity. For `entity.Incident`, there is `typekey.Incident`, which has typecodes like `TripIncident`, `InjuryIncident`, and so on.

There is a many-to-one relationship between `ExposureType` and the `Incident` typelist. Every instance of `Exposure` refers to an instance of `Incident`, and this relationship determines the specific type of incident that it is, based on the exposure's `ExposureType`. There are differences between this relationship and the others in the core LOB model:

- The categories representing this relationship are unidirectional, going only from `ExposureType` to `Incident`.
- Incident types do not appear as children of `ExposureTypes` in the LOB editor. They are non-LOB typecodes edited by using the `Other Categories` folders in the LOB editor.

See also

- “Relationships Between LOB Typelists and Other Typelists” on page 482
- “Adding a Non-LOB Typecode” on page 487

Relationships Among LOB Typelists

LOB typelists are related to each other and refer to each other by using two-way category references. A typelist category is a reference from one typecode to another typecode, usually in a different typelist. Categories allow

the typecodes in the referring typelist to be categorized into groups identified by the typecodes in the referred-to typelist. The relationship between a parent and child LOB typecode is represented by a category on the parent referring to the child and another on the child referring to the parent.

To view and edit these two-way relationships, adding a child to or removing a child from its parent, use the **LOB** tab of the Typelists editor. The **LOB** tab manages parent and child relationships among LOB typelists by changing both adjacent typelists. For information on opening and using the Typelists editor, see “Editing LOB Typelists and Typecodes” on page 484.

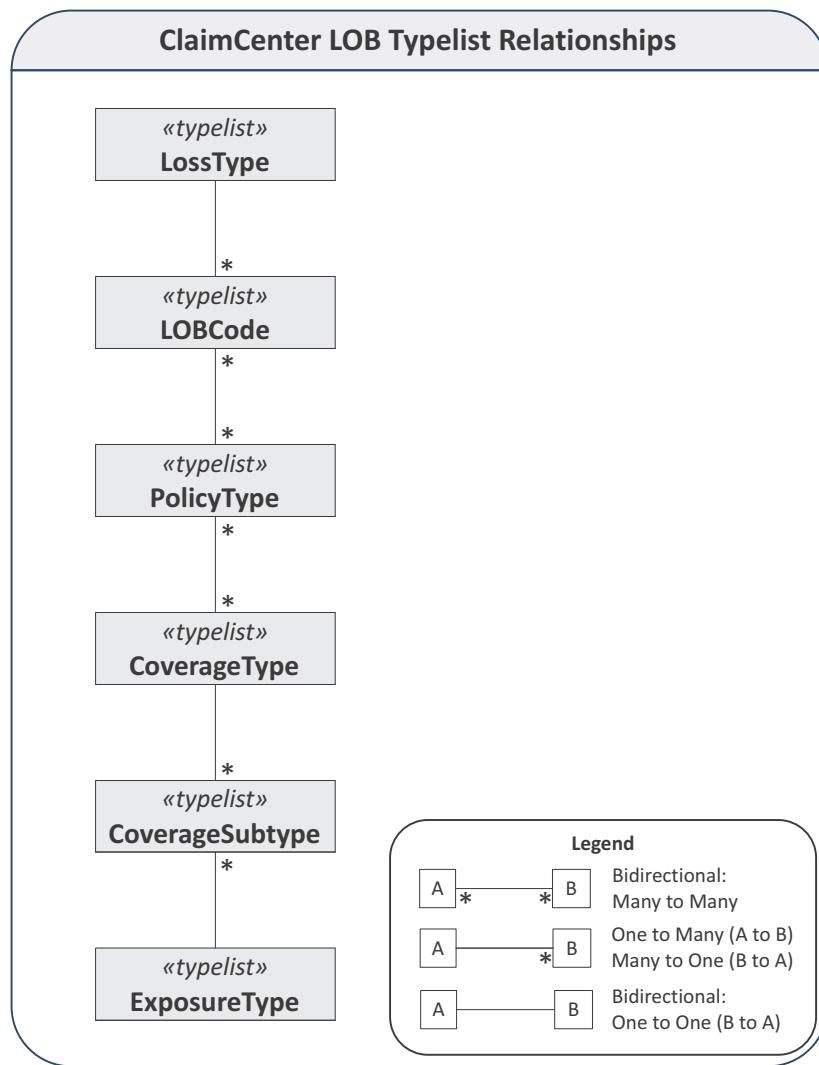
The following table lists the LOB typelists and their parent-child relationships.

TypeList	Parent	Child
LossType	-	LOBCode
LOBCode	LossType	PolicyType
PolicyType	LOBCode	CoverageType
CoverageType	PolicyType	CoverageSubtype
CoverageSubtype	CoverageType	ExposureType
ExposureType	CoverageSubtype	-

Studio represents the parent-child relationship directly in the **LOB** tab of the Typelists editor. In the tree under a typecode, you can see **Parents** and **Children** nodes for each typecode that has them.

The LOB hierarchy is not a strict hierarchy. Some relationships in the structure are many-to-many, so some typecodes can have multiple parents. For example, typecodes in **PolicyType** and **CoverageType** can have multiple parents. In contrast, typecodes in **CoverageSubtype** and **LOBCode** can have only one parent. Additionally, **LossType** has no parents, and **ExposureType** has no children.

The following diagram shows the relationships between the LOB typelists:



If a typecode has multiple parents, it is listed in the Typelists editor tree in the **Children** folder for each parent. Each of these elements represents the same typecode. Therefore, editing the typecode in one place affects all appearances of it.

For example:

1. Open the LOBCode typelist in the typelist editor and click the LOB tab.
For a description of how to open a typelist, see “Editing LOB Typelists and Typecodes” on page 484.
2. Right-click Commercial Auto Line and chose **Add new → PolicyType**.
3. For **Code** enter additionalpip. For **Name** enter Additional PIP. For **Description** enter Additional PIP.
4. Right-click the Personal Auto Line LOB code and chose **Add new → PolicyType**.
5. In the **Add PolicyType** dialog, click the **Select typecode** tab and choose the **PolicyType** you created previously, additionalpip. This typecode now becomes a child of both Commercial Auto Line and Personal Auto Line.

6. Navigate to **Personal Auto Line** → **Children** → **Additional PIP**.
7. Right-click **Additional PIP** and choose **Add new** → **CoverageType**.
8. In the **Add PolicyType** dialog, click the **Select typecode** tab and choose any **CoverageType**.
9. Navigate to **Commercial Auto Line** → **Children** → **Additional PIP** → **Children**. You can see that the **CoverageType** you added under Personal Auto Line appears there as well. Because the two Additional PIP nodes represent the same **PolicyType** typecode, their properties, including their set of parents and children, are always identical.

See also

- “LOB Typelists” on page 478
- “Relationships Between LOB Typelists and Other Typelists” on page 482
- “Editing LOB Typelists and Typecodes” on page 484
- “Cov erages and Policies” on page 489
- “Adding a New LossType Typecode” on page 491
- “Adding a New ExposureType Typecode” on page 492

Relationships Between LOB Typelists and Other Typelists

An LOB typecode can be referenced through typelist categories by a typecode from a non-LOB typelist, and it can reference a typecode in a non-LOB typelist. Outside the LOB structure, Studio does not maintain referential integrity.

If you modify one of the LOB typelists in the Typelists editor, you might also need to modify other non-LOB typelists or PCF files that reference your modified typelist. The **LOB** tab of the Typelists editor does not edit non-LOB typelists associated with these six typelists. It also does not edit Gosu code or PCF files that reference these six typelists. If you make changes to an LOB typelist that affect related typelists, Gosu code, and PCF files, you must update these related files, or you can cause errors.

This topic includes:

- “LOB Typelists Referenced by Non-LOB Typelists” on page 483
- “LOB Typelists that Reference Non-LOB Typelists” on page 483
- “Modal PCF Files that Use LOB Typelists” on page 484

See also

- “LOB Typelists” on page 478
- “Relationships Among LOB Typelists” on page 479
- “Editing LOB Typelists and Typecodes” on page 484
- “Cov erages and Policies” on page 489
- “Adding a New LossType Typecode” on page 491
- “Adding a New ExposureType Typecode” on page 492

LOB Typelists Referenced by Non-LOB Typelists

The following typelists are examples of LOB typelists that are referenced by—have incoming categories for—non-LOB typelists:

LOB typelist	Incoming categories from non-LOB typelists
CoverageSubtype	LossPartyType
CoverageType	CostCategory CovTermPattern LineCategory
PolicyType	ClaimTier ExposureTier InsuranceLine
LossType	ClaimantType ClaimSegment LossCause MetroReportType OfficialType PriContributingFactors QuickClaimDefault ResolutionType SeverityType

You can see the incoming categories for each typecode of the current typelist by clicking the Incoming Categories tab and clicking the typecode. Under it, you see the list of all typelists that reference the typecode, including the LOB typelists.

Cascading Non-LOB Typelist References

Non-LOB typelists can reference the non-LOB typelists that reference LOB typelists. These cascading references can involve more than one level of non-LOB typelists. For example:

- `LossType` is referenced by `PriContributingFactors`, which is referenced by `SecContributingFactors`, which is referenced by `ResContributingFactors`.
- `LossType` is referenced by `LossCause`, which is referenced by `AccidentType`.
- `PolicyType` is referenced by `InsuranceLine`, which is referenced by both `InsuranceSubline` and `MajorPerils`.
- `CoverageSubtype` is referenced by `LossPartyType`, which is referenced by `VehicleType`.
- `CoverageSubtype` is referenced by `CostCategory`, which is referenced by `LineCategory`.

In these cases, a change to one LOB typelist could require changes to a non-LOB typelist, which in turn requires changes to another non-LOB typelist.

LOB Typelists that Reference Non-LOB Typelists

The following typelists are examples of LOB typelists that reference—have outgoing categories for—non-LOB typelists:

LOB typelist	Outgoing categories to non-LOB typelists
CoverageSubtype	SourceSystem
ExposureType	Incident
LOBCode	SourceSystem
PolicyType	InternalPolicyType PolicyTab SourceSystem

On the **LOB** tab, you can see the non-LOB typelists that each typecode references by selecting the typecode and then opening its **Other Categories** folder.

See also

- “LOB Typelists and Incidents” on page 479

Modal PCF Files that Use LOB Typelists

A number of modal PCF files use the **LossType** and **ExposureType** typelists. In these PCF files, the detail or list view that displays is based on the **LossType** or **ExposureType** typecode. For example:

TypeList	PCF
LossType	ClaimEvaluationDetailsDV ClaimPolicyGeneral FNOLWizard_AssignSaveScreen FNOLWizard_BasicInfoScreen FNOLWizard_NewLossDetailsScreen FNOLWizard_ServicesScreen LossDetailsDV NewClaimLossDetailsDV NewClaimPolicyGeneralPanelSet PolicyGeneralPanelSet QuickClaimDV
ExposureType	NewClaimExposureDV ExposureDetailDV NewExposureDV

Editing LOB Typelists and Typecodes

To work with lines of business, you navigate in the Project window to **configuration** → **config** → **Extensions** → **TypeList** and double-click one of the following typelists:

<u>LossType.ttx</u>
<u>LOBCode.ttx</u>
<u>PolicyType.ttx</u>
<u>CoverageType.ttx</u>
<u>CoverageSubtype.ttx</u>
<u>ExposureType.ttx</u>

You can also search for and open typelists by pressing **Ctrl+Shift+N**, typing the name of the typelist, and double-clicking the **.ttx** version of the typelist in the search results.

Double-clicking an LOB typelist opens the Typelists editor and automatically selects the **LOB** tab for the tree on the left side of the editor. This tab of the editor is where you do most of your work on LOB typelists. To work in this editor, you can click a typecode or typelist node to edit its properties. You can also right-click a node to do things like adding new typecodes and children of typecodes.

Because the **LOB** tab shows the current typelist’s typecodes and all the LOB descendants, you can perform most LOB operations by editing the top-level LOB typelist, **LossType**. You can view the entire LOB structure by expanding successive folders of children.

This topic includes:

- “LOB Typelists Editor Tabs” on page 485
- “LOB Typelists Editor Right-click Menu” on page 485
- “Editing an LOB Typelist” on page 486
- “Editing an LOB Typecode” on page 486

- “Adding a Child LOB Typecode” on page 487
- “Adding a Non-LOB Typecode” on page 487
- “Adding a Parent to an LOB Typecode” on page 488
- “Removing an LOB Typecode” on page 488
- “Retiring an LOB Typecode” on page 488
- “Removing an LOB Typecode from Its Parent” on page 488
- “Exporting an LOB Typecode or Typelist” on page 489
- “Localizing an LOB Typelist” on page 489

See also

- “LOB Typelists” on page 478
- “Relationships Among LOB Typelists” on page 479
- “Relationships Between LOB Typelists and Other Typelists” on page 482
- “Cov erages and Policies” on page 489
- “Adding a New LossType Typecode” on page 491
- “Adding a New ExposureType Typecode” on page 492

LOB Typelists Editor Tabs

When you open an LOB typelist, the tree list has the following tabs at the bottom that show different types of information and support editing in different ways:

- **LOB** – A navigable tree of LOB typecodes that shows the relationships to typecodes in the other LOB typelists and maintains those relationships. This tab shows LOB typecodes with their typelist names, like `LOBCode`, `PolicyType`, and `CoverageType`. When you select a typecode, its `Parents` and `Children` folders show the hierarchical relationships to typecodes from the LOB typelists. The `Other Categories` folder shows outgoing typecode references from this typecode to typecodes in non-LOB typelists.

Note: Use the **LOB** tab when working with LOB typelists. This tab updates parent and child typecodes as necessary when you edit existing typecodes.

- **Typelist** – The current typelist simply as a tree list of the typecodes in the current typelist, along with any other elements present in the typelist XML files.
- **Incoming Categories** – Typecodes that reference typecodes in this typelist. This tab is useful for identifying typelists that refer to typecodes in the current typelist. It includes both LOB and non-LOB typelists.
- **Outgoing Categories** – Typecodes that are referenced from this typelist. This information is also available under the LOB and Typelist tabs. However, this view is organized by the typecodes that are being referred to, rather than by the typecodes in this typelist. This tab is useful for identifying typelists that a typecode references. It includes both LOB and non-LOB typelists.

Note: If you select the **LOB** tab and navigate to a typecode’s `Other Categories` folder, you see all the non-LOB typelists that the typecode references. Because this folder shows only non-LOB outgoing typelist references, it can be more useful than the **Outgoing Categories** tab. That tab shows all typelist references, including LOB typelist references. See “Relationships Between LOB Typelists and Other Typelists” on page 482.

- **XML** – The typelist as XML. This view is read-only.

LOB Typelists Editor Right-click Menu

Right-clicking a node in the tree list of the Typelists editor opens a drop-down menu of editing commands specific to the node. For example, you can right-click an `LOBCode` typecode and do things like adding a new `PolicyType` child or removing the selected typecode from its parent. Or, in that same typelist, you can right-click the root node of the tree, which represents the typelist itself, and add a new `LOBCode` typecode to the typelist.

The right-click menu provides the following commands:

- **Add new** – Add a new typecode, category, or category list. See “Adding a Child LOB Typecode” on page 487.
 - *LOB-typelist-name* – Add a typecode of the typelist *LOB-typelist-name*. You see this command only if you can add a child to the selected node. The editor displays the appropriate typelist for the node you right-click. For example, if you are editing the **CoverageType** typelist and you click the top node in the tree, **CoverageType**, you can add a **CoverageType** typecode to this typelist. In the same tree, if you select any **CoverageType** typecode, such as **Baggage**, you can add a **CoverageSubtype** typecode as a child.
 - **Category** – Add a category to refer to a typecode that is not an LOB typelist child or parent of this typecode. This command is useful for adding a reference to a non-LOB typelist.
 - **Categorylist** – This command associates this typecode with all the typecodes in the specified typelist. It does not affect the LOB relationships. Category lists are described in “Dynamic Filters” on page 287.
- **Duplicate** – Create a copy of the selected typecode. The copy has a number appended to its code and name that distinguishes it from the original. This command is not available for the root node, which represents the typelist itself.
- **Remove** – This command can cause errors for objects that reference the typecode. Guidewire recommends that instead of removing a typecode, you change its **retired** property to **true**. This command is not available for the root node, which represents the typelist itself. See “Removing an LOB Typecode” on page 488.
- **Remove from parent** – Remove the typecode from its parent. You must be editing the parent typelist or another typelist higher in the LOB hierarchy, and then navigate from the parent to the child to see this menu command. See “Removing an LOB Typecode from Its Parent” on page 488.
- **Cut, Copy, Paste** – You do not typically use these commands when working with LOB typelists.
- **Export Branch** – Exports the tree to a CSV file or an HTML file. The currently selected node is the root of the exported LOB hierarchy tree. See “Exporting an LOB Typecode or Typelist” on page 489.

Editing an LOB Typelist

For information on opening an LOB typelist for editing, see “Editing LOB Typelists and Typecodes” on page 484. The top level LOB typelist is **LossType.ttx**. You can open this typelist to get a top-down view of all the associated LOB typelists. Typically, you open the typelist you want to work with. For example, if you want to remove a coverage type from one of its parent policy types, you could open **CoverageType.ttx** to make the change.

If you want to edit properties of the typelist itself, open that typelist and click the root node, which represents the typelist. The only practical edit you can make to the properties is to change the **desc** property.

If you right-click the root node, which represents the typelist itself, you can click **Add new** to add new typecodes to the typelist. There are additional commands on this right-click menu, described at “LOB Typelists Editor Right-click Menu” on page 485.

See also

“Adding a Child LOB Typecode” on page 487.

Editing an LOB Typecode

To edit typecodes of an LOB typelist, open the typelist in the editor and click the **LOB** tab. You can navigate up and down the tree in the **LOB** tab, editing parents and children of the current typecode. Your changes affect all the related LOB typelists. You can select a typecode and edit its properties, or you can right-click a typecode and add a new child, a new category, or a new category list.

Note: For information on opening an LOB typelist for editing, see “Editing LOB Typelists and Typecodes” on page 484.

If you select a typecode in the tree, you see the properties of the typecode on the right, and you can edit them. For example, you can edit a typecode's **code**, **name**, **desc**, **priority**, and **retired** fields.

- If you edit the **name** property, Studio updates the typecode name in the tree.
- If you edit the **code** property, Studio updates any related typecodes in LOB typelists, such as parent and children typecodes. However, you must manually update any non-LOB typecodes that reference this one or that this one references.
- If you select **retired**, the typecode is no longer available in the user interface, nor is it available to new instances of objects in the ClaimCenter server. However, existing objects, such as existing claims, that already reference the typecode can continue to do so. Setting this property is almost always preferable to removing the typecode.

Adding a Child LOB Typecode

Use the **Add new** right-click menu command to add a new child typecode to a typecode in a typelist.

Note:

You can also use **Add new** to add a category or category list and to add a typecode to a typelist. See:

- “LOB Typelists Editor Right-click Menu” on page 485
- “Editing an LOB Typelist” on page 486
- “Adding a Non-LOB Typecode” on page 487

This menu command is available for a typecode only if you can add additional children to it. For example, the parent-child relationship for some typelists is one-to-one, and if there is already a child, you cannot add another. For **ExposureType**, no children are supported. See “Relationships Among LOB Typelists” on page 479.

Clicking **Add new** enables you to add the child typecode appropriate for the selected typecode, such as adding a child **LOBCode** to **LossType** or adding a child **CoverageType** to **PolicyType**. For example, if you right-click one of the **LossType** typecodes, such as **Auto**, clicking **Add new** → **LOBCode** enables you to add a child **LOBCode** typecode. This command opens a dialog that you use to select an existing typecode from a typelist or enter values for a new typecode. In the dialog, click either the **New typecode** tab to add a new typecode or the **Select typecode** tab to add an existing typecode.

If there are existing child elements in the child typelist, the **Select typecode** tab is available in the dialog. If there are no more existing child typecodes, the **Select typecode** tab is disabled. In this case, you must use the **New typecode** tab to define a new typecode.

When you add a new child, the editor creates a new typecode in the typelist in the next level down, and it sets up the parent-child relationship. A new typecode cannot have the same code as an existing typecode in the same typelist. Studio checks for this condition and prevents you from performing this action in the editor.

IMPORTANT As you add LOB typecodes, pay close attention to any errors you might receive. For example, if you add a new **LOBCode** typecode to the **LOBCode** typelist, you see an error saying that the typecode must have at least one category from typelist **PolicyType**. If you ignore this error, the ClaimCenter server will not start until you correct it.

Adding a Non-LOB Typecode

As described in “Relationships Between LOB Typelists and Other Typelists” on page 482, an LOB typecode can reference a typecode in a non-LOB typelist. To add a reference to a non-LOB typecode:

1. Open an LOB typelist for editing. See “Editing LOB Typelists and Typecodes” on page 484.
2. Right-click the typecode to which you want to add the reference and choose **Add new** → **Category**.

The editor opens the typecode's **Other Categories** folder and adds a new node to it named `<empty>`. You also see an error. The error will go away after you enter the code and typelist of the new typecode.

3. Click the new `<empty>` node so you can edit its properties.
4. Enter the typelist and the typecode's **code** for the non-LOB typelist. If you begin typing a name in the **typelist** field, the list changes to narrow your choices. If you first select the typelist name from the list of typelists, you can then use the **code** drop-down list to select the code.

When you set the properties for the non-LOB typelist, the Typelists editor changes the name of the node and the errors go away.

Adding a Parent to an LOB Typecode

You can add a parent to an LOB typecode that supports one or more parents by opening the parent typelist and adding the typecode as a child. See “Adding a Child LOB Typecode” on page 487.

Removing an LOB Typecode

While the **Remove** command is available for typecodes in a typelist, Guidewire recommends that you not remove a typecode. Existing objects that reference the typecode, such as claims, can become invalid as a result. Additionally, removing a typecode can make typecodes it references into orphans if those typecodes have no other parents.

A circumstance in which you can safely remove a typecode is during development of new typecodes that have not gone into production. Even in that case, you must clean up all references to the typecode in other typelists, PCF files, and Gosu code.

Instead of removing a typecode, you can retire it or remove it from its parent. See:

- “Retiring an LOB Typecode” on page 488
- “Removing an LOB Typecode from Its Parent” on page 488

Retiring an LOB Typecode

If you no longer want to use a typecode, set its **retired** property to **true** as described in “Editing an LOB Typelist” on page 486. When you retire a typecode, it is no longer used by any PCF files or Gosu code that reference it. Retiring a typecode is the preferred way to cleanly remove entire policy types, coverages, coverage subtypes, and so on.

Removing an LOB Typecode from Its Parent

You can remove a child LOB typecode from its parent typelist. You typically do so because you want to remove the functionality the typecode represents from that particular parent typelist. Typically, both the child and the parent would have multiple children. Removing the typecode from its parent does not remove the typecode itself—it just removes the reference to the child from the parent.

Note: If you want to actually remove a typecode, retire it. See “Retiring an LOB Typecode” on page 488.

Use the right-click menu command **Remove typecode from parent** to remove a typecode from its parent. To see this menu command, you must first be editing a typelist that is higher in the LOB hierarchy than the typecode you want to remove from its parent. With the parent typelist or higher level typelist open in the editor, navigate from the parent LOB typecode to the child typecode. Then right-click the child typecode. For a hierarchical list of typelists, see “Relationships Among LOB Typelists” on page 479.

For example, you want to remove Arkansas personal injury protection coverage from your business auto policy type, but not from your personal auto policy type. You remove the `CoverageType` typecode `PIP - Arkansas`, code `PAPIP_AR`, from its parent `PolicyType` typecode `Commercial Auto`, code `BusinessAuto`, as follows:

1. Navigate to `configuration` → `config` → `Extensions` → `Typelist` and double-click `PolicyType.ttx`.
2. In the Typelists editor, navigate to `Commercial Auto` → `Children` → `PIP - Arkansas`.
3. Right-click `PIP - Arkansas`, and then, in the drop-down menu, click `Remove typecode from parent`.

The coverage is removed from Business Auto, but remains a coverage for Personal Auto. The coverage type therefore still has a parent. It also continues to be a typecode in the `CoverageType` typelist.

Note: This example removes a typecode from one of two parents. For a typecode with only one parent, such as `Bobtail Liability`, removing it from its parent would orphan the typecode, leaving it without a parent in the `CoverageType` typelist.

Exporting an LOB Typecode or Typelist

You can right-click a node of an LOB typelist and choose `Export Branch` to export the current typecode or typelist and the entire LOB tree below it to a file. You can select whether to export the selected LOB typecode hierarchy as HTML or as CSV. These two commands, available at all levels, export the tree either to an HTML file or to a CSV file. Studio exports the tree in a tabular format. You can print the tree or work with it as a spreadsheet, as follows:

- To print the tree, export it to HTML and use a web browser to print it out.
- To use the tree in a spreadsheet editor like Microsoft Excel, export it to CSV.

Localizing an LOB Typelist

To see the US English localized LOB typecodes, navigate to `configuration` → `config` → `Localizations` → `en_US` and double-click `typelist.properties`. For more information on localizing typecodes, see “Typecodes” on page 36 in the *Globalization Guide*.

Coverages and Policies

After you obtain a policy snapshot or refresh, ClaimCenter transfers coverages from the policy. However, coverages in a policy are typically more detailed than is needed in ClaimCenter. For example, a policy holder can have several coverages that apply to property damage. In the base configuration, the snapshot contains only one of these coverages. However, you can configure it to match your business requirements.

Also, the actual aggregate limit for a coverage can be lower than the policy’s total aggregate limit. Only the coverage limit of the coverage used applies to the aggregate limit.

The coverages used are typecodes in LOB typelists.

Personal Auto Coverages and the LOB Typelists

This topic uses as an example one set of coverage types in the base configuration. For this example, open the `LossType` typelist in the `LOB` tab of the Typelists editor, click the `Auto` loss type and then click `Children`. You can see

elements of the Personal Auto Line line of business when you expand it and its child, the Personal Auto policy type. The following figure shows the values you see in the first line of the table of coverages.

LOB	Typecode	TypeList	Name	Value
LossType	AUTO	LossType	code	PersonalAutoLine
Auto		LOBCode	name	Personal Auto Line
Children		BusinessAutoLine	desc	Personal Auto
Commercial Auto Line			priority	-1
Children		LOBCode		
Parents		PolicyType		
Other Categories		LossType		
Personal Auto Line	PersonalAutoLine	LOBCode	retired	<input type="checkbox"/> false
Children		PolicyType		
Personal Auto	PersonalAuto	PolicyType		
Children		CoverageType		
Collision	PACollisionCov	CoverageType		
Children		CoverageSubtype		
Collision	PACollisionCov	CoverageSubtype		
Children		ExposureType		
Vehicle	VehicleDamage	ExposureType		

The following table shows the CoverageType, CoverageSubtype, and ExposureType typecodes associated with the Personal Auto policy type in the ClaimCenter base configuration:

Coverage type	Coverage subtype (Children)	Exposure type (Children)
Collision (PACollisionCov)	Collision (PACollisionCov)	Vehicle (VehicleDamage)
Collision - Limited Coverage (PACollision_MA_MI_Limited)	Collision - Limited Coverage (PACollision_MA_MI_Limited)	Vehicle (VehicleDamage)
Comprehensive (PAComprehensiveCov)	Comprehensive (PAComprehensiveCov)	Vehicle (VehicleDamage)
Death and Disability Benefit (PADeathDisabilityCov)	Death and Disability Benefit (PADeathDisabilityCov)	Bodily Injury (BodilyInjuryDamage)
Electronic Equipment (PAExcessElectronicsCov)	Electronic Equipment (PAExcessElectronicsCov)	Vehicle (VehicleDamage)
Liability - Bodily Injury and Property Damage (PALiabilityCov)	Liability - Bodily Injury (PALiabilityCov_bi)	Bodily Injury (BodilyInjuryDamage)
	Liability - PropertyDamage (PALiabilityCov_pd)	Property (PropertyDamage)
	Liability - Vehicle Damage (PALiabilityCov_vd)	Vehicle (VehicleDamage)
Medical Payments (PAMedPayCov)	Medical Payments (PAMedPayCov)	Med Pay (MedPay)
Mexico Coverage - Limited (PALimitedMexicoCov)	Mexico Cov - BI (PAMexicoCovBI)	Bodily Injury (BodilyInjuryDamage)
	Mexico Cov - Gen. Damages (PAMexicoCovGEN)	General (GeneralDamage)
	Mexico Cov - PD (PAMexicoCovPD)	Property (PropertyDamage)
	Mexico Cov - Vehicle Damage (PAMexicoCovVEH)	Vehicle (VehicleDamage)
PIP - Arkansas (PAPIP_AR)	PIP - Arkansas (PAPIP_AR)	PIP (PIPDamages)

Coverage type	Coverage subtype (Children)	Exposure type (Children)
PIP - Delaware (PAPIP_DE)	PIP - Delaware (PAPIP_DE)	PIP (PIPDamages)
PIP - District of Columbia (PAPIP_DC)	PIP - District of Columbia (PAPIP_DC)	PIP (PIPDamages)
PIP - Florida (PAPIP_FL)	PIP - Florida (PAPIP_FL)	PIP (PIPDamages)
PIP - Hawaii (PAPIP_HI)	PIP - Hawaii (PAPIP_HI)	PIP (PIPDamages)
PIP - Kansas (PAPIP_KS)	PIP - Kansas (PAPIP_KS)	PIP (PIPDamages)
PIP - Kentucky (PAPIP_KY)	PIP - Kentucky (PAPIP_KY)	PIP (PIPDamages)
PIP - Maryland (PAPIP_MD)	PIP - Maryland (PAPIP_MD)	PIP (PIPDamages)
PIP - Massachusetts (PAPIP_MA)	PIP - Massachusetts (PAPIP_MA)	PIP (PIPDamages)
PIP - Michigan (PAPIP_MI)	PIP - Michigan (PAPIP_MI)	PIP (PIPDamages)
PIP - Minnesota (PAPIP_MN)	PIP - Minnesota (PAPIP_MN)	PIP (PIPDamages)
PIP - New Jersey (PAPIP_NJ)	PIP - New Jersey (PAPIP_NJ)	PIP (PIPDamages)
PIP - New York (PAPIP_NY)	PIP - New York (PAPIP_NY)	PIP (PIPDamages)
PIP - North Dakota (PAPIP_ND)	PIP - North Dakota (PAPIP_ND)	PIP (PIPDamages)
PIP - Oregon (PAPIP_OR)	PIP - Oregon (PAPIP_OR)	PIP (PIPDamages)
PIP - Pennsylvania (PAPIP_PA)	PIP - Pennsylvania (PAPIP_PA)	PIP (PIPDamages)
PIP - Texas (PAPIP_TX)	PIP - Texas (PAPIP_TX)	PIP (PIPDamages)
PIP - Utah (PAPIP_UT)	PIP - Utah (PAPIP_UT)	PIP (PIPDamages)
PIP - Washington (PAPIP_WA)	PIP - Washington (PAPIP_WA)	PIP (PIPDamages)
Property Protection Insurance (PAPropProtectionCov)	Property Protection Insurance (PAPropProtectionCov)	Property (PropertyDamage)
Rental Car Loss of Use (PALossOfUseCov)	Rental Car Loss of Use (PALossOfUseCov)	Loss Of Use (LossOfUseDamage)
Rental Reimbursement (PARentalCov)	Rental Reimbursement (PARentalCov)	Vehicle (VehicleDamage)
Tape / Disc Media (PATapeDiscMediaCov)	Tape / Disc Media (PATapeDiscMediaCov)	Personal Property (PersonalPropertyDamage)
Towing and Labor (PATowingLaborCov)	Towing and Labor (PATowingLaborCov)	Towing and Labor (TowOnly)
Underinsured Motorist - Bodily Injury (PAUIMBICov)	Underinsured Motorist - Bodily Injury (PAUIMBICov)	Bodily Injury (BodilyInjuryDamage)
Underinsured Motorist - Property Damage (PAUIMPDCov)	Underinsured Motorist - VehicleDamage (PAUIMPDCov)	Vehicle (VehicleDamage)
Uninsured Motorist - Bodily Injury (PAUMBICov)	Uninsured Motorist - Bodily Injury (PAUMBICov)	Bodily Injury (BodilyInjuryDamage)
Uninsured Motorist - Property Damage (PAUMPDCov)	Uninsured Motorist - Property Damage (PAUMPDCov)	Vehicle (VehicleDamage)

Adding a New LossType Typecode

For each new loss type typecode that you define, you must also do the following:

1. Add the typecode to `LossType` typelist. See “Editing an LOB Typelist” on page 486.
2. Add an `LOBCode` child typecode to the new typecode. See “Adding a Child LOB Typecode” on page 487.
3. Update non-LOB typelists that reference `LossType` typecodes. These typelists include `ClaimantType` and `LossCause`. See “LOB Typelists Referenced by Non-LOB Typelists” on page 483.
4. Create the supporting detail and list views in the ClaimCenter interface.

Creating the Detail Views and List Views for the Loss Type

You define both the detail views and panel sets in individual PCF files. Claim-related and policy-related screens in ClaimCenter can then reference these files. The PCF files that you need to add are modal. With modal PCF files, the screen file that references these pages calls the appropriate detail or list view based on the `LossType` typecode.

It is important that your PCF files follow a naming convention that includes the loss type code in the file name. Use the following naming convention, replacing `Code` with the actual typecode:

`FileName.Code`

You must capitalize the first character of the included typecode. For example, a typecode of `Cargo` would have the following corresponding file name:

`NewClaimWizardLossDetailsDV.Cargo`

For each typecode that you add, create the following claim-related files. In the Project window, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **claim**. Then open the folder for each file in the table that follows to add the new file. Replace `Code` with the name of your typecode.

Claim folder	File	Supports
<code>lossdetails</code>	<code>LossDetailsDV.Code</code>	Editing existing claims
<code>FNOL</code>	<code>NewClaimLossDetailsDV.Code</code> <code>NewClaimPolicyGeneralPanelSet.Code</code> <code>QuickClaimDV.Code</code>	New Claim wizard
<code>planofaction</code>	<code>ClaimEvaluationDetailsDV.Code</code>	Creating and editing claim evaluations
<code>policy</code>	<code>PolicyGeneralPanelSet.Code</code> <code>PolicySummaryGeneralDV.Code</code>	Policy edits and searches
<code>snapshot → version</code>	<code>ClaimSnapshotGeneralVersionDV.Code</code> <code>ClaimSnapshotGeneralVersionPanelSet.Code</code>	Claim snapshots

See also

- “LOB Typelists” on page 478
- “Relationships Among LOB Typelists” on page 479
- “Relationships Between LOB Typelists and Other Typelists” on page 482
- “Editing LOB Typelists and Typecodes” on page 484

Adding a New ExposureType Typecode

For each new exposure type that you define, you must also do the following:

1. Add the new typecode to the `ExposureType` typelist. See “Editing an LOB Typelist” on page 486.
2. Add a parent `CoverageSubtype` to the new `ExposureType` typecode. See “Adding a Child LOB Typecode” on page 487.
3. Create the supporting detail views in the ClaimCenter interface.

Creating the Necessary Detail Views for the Exposure Type

You define the detail views in individual PCF files. Claim-related and exposure-related screens in ClaimCenter can then reference these files. The PCF files that you need to add are modal. With modal PCF files, the screen file that references these pages calls the appropriate detail view based on the `ExposureType` typecode.

It is important that your PCF files follow a naming convention that includes the exposure typecode in the file name. Use the following naming convention, replacing *Code* with the actual typecode:

`FileName.Code`

You must capitalize the first character of the included typecode. For example, a typecode of *CargoDamage* would have a corresponding file named:

`ExposureDetailDV.Cargodamage`

For each typecode that you add, create the following claim-related and exposure-related files. In the Project window, navigate to configuration → config → Page Configuration → pcf → claim. Then open the folder for each file in the table that follows to add the new file. Replace *Code* with the name of your typecode.

Claim folder	File	Supports
exposures	<code>ExposureDetailDV.Code</code>	Editing existing exposures
FNOL	<code>NewClaimExposureDV.Code</code>	New Claim wizard
newexposure	<code>NewExposureDV.Code</code>	New Exposure wizard
<code>snapshot → version</code>	<code>ClaimSnapshotExposureversionDV.Code</code>	Claim snapshots

See also

- “LOB Typelists” on page 478
- “Relationships Among LOB Typelists” on page 479
- “Relationships Between LOB Typelists and Other Typelists” on page 482
- “Editing LOB Typelists and Typecodes” on page 484



chapter 36

Configuring Services

The Services feature in ClaimCenter 8.0 enables the creation, submission, and management of service requests in collaboration with selected vendors. ClaimCenter uses a contact management system like ContactManager to access and select vendors specializing in specific services and a vendor portal like the Guidewire Vendor Portal to communicate with vendors.

Note: In this topic and included examples, Guidewire ContactManager is used as the default contact management system, and the Guidewire Vendor Portal is used as the default vendor communication portal. If you use non-standard components to manage contacts and vendors, please ensure they are integrated appropriately with Guidewire ClaimCenter before proceeding with adding and managing services.

This topic describes the configuration of service requests and includes:

- “Importing Services” on page 495
- “Service Request Data Model” on page 502
- “Lifecycle of a Service Request” on page 503
- “Configuring Service Metrics” on page 507

See also

- “Services” on page 375 in the *Application Guide*

Importing Services

A *service* can be defined as any action that can be requested from a third-party vendor or internal provider. A service request summarizes the instruction sent to the vendor detailing the service to be performed. Each service request is associated with one claim.

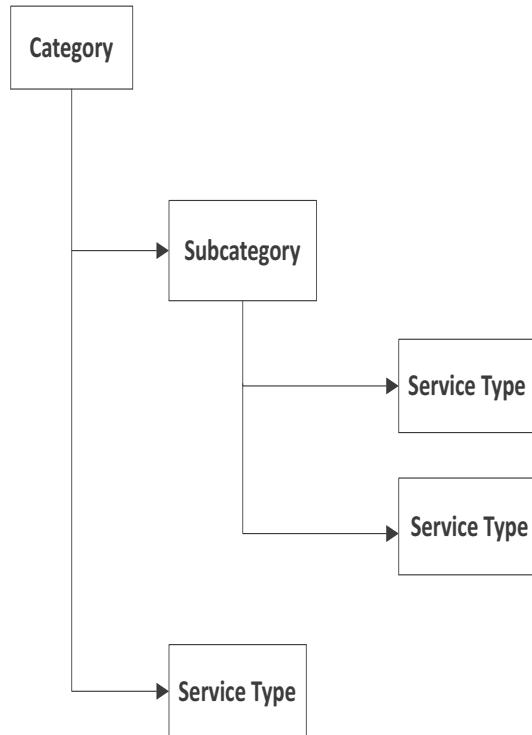
Vendor information is presented in ClaimCenter in the form of a vendor directory, from which you can make a selection based on the type of service required. The services vendor directory is organized in a tree structure, where folder nodes represent service categories and leaf nodes represent specialized services. The services performed by a specialist vendor can be configured in Guidewire Contact Manager.

Note: Guidewire recommends that you install Guidewire Contact Manager and integrate it with Guidewire ClaimCenter before importing services information.

Vendor Service Tree

Vendor services are described in the `vendorservicetree.xml` data file. The vendor service tree is imported and used by both ClaimCenter and Contact Manager. It is assumed that both applications are integrated and will maintain identical service trees. You can accomplish this by using the data import feature in both ClaimCenter and Contact Manager to load the same, customized service tree XML file.

The structure of the vendor service tree is outlined below:



At the topmost level of the tree, the folder nodes represent service categories. Under these, you can define service subcategories or service types. The leaf nodes of the tree represent the specialized services grouped into each category.

The format of the `vendorservicetree.xml` file needs to conform to a corresponding structure. The following example file defines Auto and Property categories, along with subcategories and services.

```

<?xml version="1.0"?>
<import>
<SpecialistService public-id="svc:aut">
  <Active>true</Active>
  <Code>auto</Code>
  <Description>Auto</Description>
  <Description_L10N_ARRAY/>
  <Name>Auto</Name>
  <Name_L10N_ARRAY/>
  <Parent/>
</SpecialistService>
<SpecialistService public-id="svc:aut_ins">
  <Active>true</Active>
  <Code>autoinsprepair</Code>
  <Description>Auto - Inspect / Repair</Description>
  <Description_L10N_ARRAY/>
  <Name>Inspect / Repair</Name>
  <Name_L10N_ARRAY/>
  <Parent public-id="svc:aut"/>
</SpecialistService>
  
```

```
<SpecialistService public-id="svc:aut_ins_aud">
  <Active>true</Active>
  <Code>autoinsprepairaudio</Code>
  <Description>Auto - Inspect / Repair - Audio equipment</Description>
  <Description_L10N_ARRAY/>
  <Name>Audio equipment</Name>
  <Name_L10N_ARRAY/>
  <Parent public-id="svc:aut_ins"/>
</SpecialistService>
<SpecialistService public-id="svc:aut_ins_bod">
  <Active>true</Active>
  <Code>autoinsprepairbody</Code>
  <Description>Auto - Inspect / Repair - Auto body</Description>
  <Description_L10N_ARRAY/>
  <Name>Auto body</Name>
  <Name_L10N_ARRAY/>
  <Parent public-id="svc:aut_ins"/>
</SpecialistService>
<SpecialistService public-id="svc:pro">
  <Active>true</Active>
  <Code>prop</Code>
  <Description>Property</Description>
  <Description_L10N_ARRAY/>
  <Name>Property</Name>
  <Name_L10N_ARRAY/>
  <Parent/>
</SpecialistService>
<SpecialistService public-id="svc:pro_ins">
  <Active>true</Active>
  <Code>propinspect</Code>
  <Description>Property - Inspection</Description>
  <Description_L10N_ARRAY/>
  <Name>Inspection</Name>
  <Name_L10N_ARRAY/>
  <Parent public-id="svc:pro"/>
</SpecialistService>
<SpecialistService public-id="svc:pro_ins_ind">
  <Active>true</Active>
  <Code>propinspectindependent</Code>
  <Description>Property - Inspection - Independent appraisal</Description>
  <Description_L10N_ARRAY/>
  <Name>Independent appraisal</Name>
  <Name_L10N_ARRAY/>
  <Parent public-id="svc:pro_ins"/>
</SpecialistService>
<SpecialistService public-id="svc:pro_con">
  <Active>true</Active>
  <Code>propconstrserv</Code>
  <Description>Property - Construction services</Description>
  <Description_L10N_ARRAY/>
  <Name>Construction services</Name>
  <Name_L10N_ARRAY/>
  <Parent public-id="svc:pro"/>
</SpecialistService>
<SpecialistService public-id="svc:pro_con_gen">
  <Active>true</Active>
  <Code>propconstrservgencontractor</Code>
  <Description>Property - Construction services - General contractor</Description>
  <Description_L10N_ARRAY/>
  <Name>General contractor</Name>
  <Name_L10N_ARRAY/>
  <Parent public-id="svc:pro_con"/>
</SpecialistService>
<SpecialistService public-id="svc:pro_con_car">
  <Active>true</Active>
  <Code>propconstrservcarpentry</Code>
  <Description>Property - Construction services - Carpentry</Description>
  <Description_L10N_ARRAY/>
  <Name>Carpentry</Name>
  <Name_L10N_ARRAY/>
  <Parent public-id="svc:pro_con"/>
</SpecialistService>
</import>
```

In this example, **Auto** is defined as a service category, **Inspect/Repair** is a subcategory, and **Auto body** a service type. The services that can be performed by a vendor are configured in Guidewire ContactManager.

In this vendor service tree example, the following services are defined:

Category	Subcategory	Service Type
Auto	Inspect/Repair	Audio equipment
		Auto body
Property	Inspection	Independent appraisal
	Construction services	General contractor
		Carpentry

Using this structure, you can customize the vendor service hierarchy as needed, based on your business needs. When customizing the service tree, note the following:

- In the base configuration and sample data, the services are organized into categories and subcategories, but this nomenclature is specific to this configuration. You can alter the naming structure as well as the depth of the tree, which only has three levels in the base configuration.
- Note:** If you change the depth of the vendor services tree, you will need to update the corresponding PCF file, `InstructionServicesLV_default.pcf`, to show the correct number of columns.
- A leaf node can appear anywhere in the tree hierarchy, but each leaf node can have only one parent node. If a leaf node needs to be associated with more than one parent node, duplicate it so that it has, at most, one parent in each occurrence.

Vendor Service Details

In addition to the vendor service tree XML file, you need a second data file, `vendorservicedetails.xml`, for ClaimCenter. The vendor service details file associates each service with a compatible incident type and service request type.

Unlike `vendorservicetree.xml`, the service details XML file is imported only into ClaimCenter. It contains instances of entities that exist only in ClaimCenter, not ContactManager.

The `vendorservicedetails.xml` file must conform to a format like the example below:

```
<?xml version="1.0"?>
<import>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:23">
  <Kind>quoteonly</Kind>
  <Service public-id="svc:pro_ins_ind"/>
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:29">
  <Kind>quoteandservice</Kind>
  <Service public-id="svc:aut_ins_bod"/>
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:30">
  <Kind>quoteonly</Kind>
  <Service public-id="svc:aut_ins_bod"/>
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:31">
  <Kind>unmanaged</Kind>
  <Service public-id="svc:aut_ins_bod"/>
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:36">
  <Kind>quoteandservice</Kind>
  <Service public-id="svc:aut_ins_aud"/>
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:37">
  <Kind>quoteonly</Kind>
  <Service public-id="svc:aut_ins_aud"/>
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:75">
  <Kind>quoteandservice</Kind>
  <Service public-id="svc:pro_con_gen"/>
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:76">
  <Kind>quoteonly</Kind>
  <Service public-id="svc:pro_con_gen"/>
```

```
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:77">
    <Kind>quoteandservice</Kind>
    <Service public-id="svc:pro_con_car"/>
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleServiceRequestKind public-id="cc:78">
    <Kind>quoteonly</Kind>
    <Service public-id="svc:pro_con_car"/>
</SpecialistServiceCompatibleServiceRequestKind>
<SpecialistServiceCompatibleIncidentType public-id="cc:1">
    <IncidentType>Incident</IncidentType>
    <Service public-id="svc:aut"/>
</SpecialistServiceCompatibleIncidentType>
<SpecialistServiceCompatibleIncidentType public-id="cc:2">
    <IncidentType>Incident</IncidentType>
    <Service public-id="svc:pro"/>
</SpecialistServiceCompatibleIncidentType>
<SpecialistServiceCompatibleIncidentType public-id="cc:8">
    <IncidentType>MobilePropertyIncident</IncidentType>
    <Service public-id="svc:aut"/>
</SpecialistServiceCompatibleIncidentType>
<SpecialistServiceCompatibleIncidentType public-id="cc:10">
    <IncidentType>VehicleIncident</IncidentType>
    <Service public-id="svc:aut"/>
</SpecialistServiceCompatibleIncidentType>
<SpecialistServiceCompatibleIncidentType public-id="cc:12">
    <IncidentType>FixedPropertyIncident</IncidentType>
    <Service public-id="svc:pro"/>
</SpecialistServiceCompatibleIncidentType>
<SpecialistServiceCompatibleIncidentType public-id="cc:15">
    <IncidentType>PropertyIncident</IncidentType>
    <Service public-id="svc:aut"/>
</SpecialistServiceCompatibleIncidentType>
<SpecialistServiceCompatibleIncidentType public-id="cc:16">
    <IncidentType>PropertyIncident</IncidentType>
    <Service public-id="svc:pro"/>
</SpecialistServiceCompatibleIncidentType>
<SpecialistServiceCompatibleIncidentType public-id="cc:20">
    <IncidentType>OtherStructureIncident</IncidentType>
    <Service public-id="svc:pro"/>
</SpecialistServiceCompatibleIncidentType>
<SpecialistServiceCompatibleIncidentType public-id="cc:23">
    <IncidentType>DwellingIncident</IncidentType>
    <Service public-id="svc:pro"/>
</SpecialistServiceCompatibleIncidentType>
</import>
```

The Service public-id value in this file must be set to the SpecialistService public-id from the vendor service tree file. For example, an InjuryIncident type can be mapped to an Auto service and a quoteandservice request type.

When customizing the `vendorservicedetails`, the following constraints apply:

- Every root SpecialistService (a service without a parent) must have at least one linked compatible incident type.
- Every leaf SpecialistService (a service without any children) must have at least one linked compatible ServiceRequestKind.

Configuring Services – Steps

Services are imported into ClaimCenter using two predefined XML files:

- Vendor Service Tree (`vendorservicetree.xml`) – Lists the services by category. This file is used by both ClaimCenter and ContactManager.
- Vendor Service Details (`vendorservicedetails.xml`) – Associates services to incidents and service request types. This data file is only used in ClaimCenter.

In the base configuration, samples of these files are provided to you in the `config/sampleddata` directory. Use the steps below to add services:

1. Customize the sample XML files to tailor them to your business requirements.
2. Import the `vendorservicetree.xml` file into ClaimCenter and ContactManager using the **Administration** tab.

3. Import the `vendorservicedetails.xml` file only into ClaimCenter using the **Administration** tab.
 4. Once data load is complete, run database consistency checks (DBCC). See “[Checking Database Consistency](#)” on page 42 in the *System Administration Guide*.
- Note:** Ensure that ClaimCenter and ContactManager contain identical vendor service directories. In other words, use the same `vendorservicetree.xml` for both these applications.

See also

- “[Installing Sample Data](#)” on page 53 in the *Installation Guide*

Editing Services

Once the Services XML data is imported into the applications, you cannot edit it using the application user interface. Instead, edit the corresponding XML file, as needed, and import it back into ClaimCenter and ContactManager.

Note: You are recommended to always choose to overwrite changes, if prompted during import, so the service tree in your database exactly matches the imported XML file.

Deleting Services

You cannot delete a service from the vendor service tree, once it is created in the XML file. However, you can mark it as inactive and import the tree back into the applications. The service is then no longer available in the application user interface.

In the following example, the `Auto - Adjudicate claim` service is disabled.

```
<SpecialistService public-id="svc:aut_adj">
  <Active>false</Active>
  <Code>autoadjudicate</Code>
  <Description>Auto - Adjudicate claim</Description>
  <Description_L10N_ARRAY>
    <Name>Adjudicate claim</Name>
    <Name_L10N_ARRAY>
      <Parent public-id="svc:aut"/>
    </Name_L10N_ARRAY>
  </Description_L10N_ARRAY>
</SpecialistService>
```

Localizing Services

You can extend the `SpecialistService` entity with localization elements, depending on the user’s preferred language needs.

Specifying Localized Languages for a SpecialistService Instance

You can use `<Language>` elements with a `SpecialistService` entity if you require that the service have a different name or description depending on the user’s preferred language.

Note: You do not provide a language for the `<Code>` element because it is used only in code and is not visible to the user.

The following XML definition for the Auto category provides Peruvian Spanish and German alternatives for the `Name` and `Description` fields:

```
<SpecialistService public-id="svc:1">
  <Code>auto</Code>
  <Description>Auto</Description>
  <Description_L10N_ARRAY>
    <SpecialistService_Description_L10N public-id="svc:1">
      <Language>es_PE</Language>
      <Owner public-id="svc:1"/>
      <Value>Vehículo</Value>
    </SpecialistService_Description_L10N>
    <SpecialistService_Description_L10N public-id="svc:2">
      <Language>de_DE</Language>
      <Owner public-id="svc:1"/>
    </SpecialistService_Description_L10N>
  </Description_L10N_ARRAY>
</SpecialistService>
```

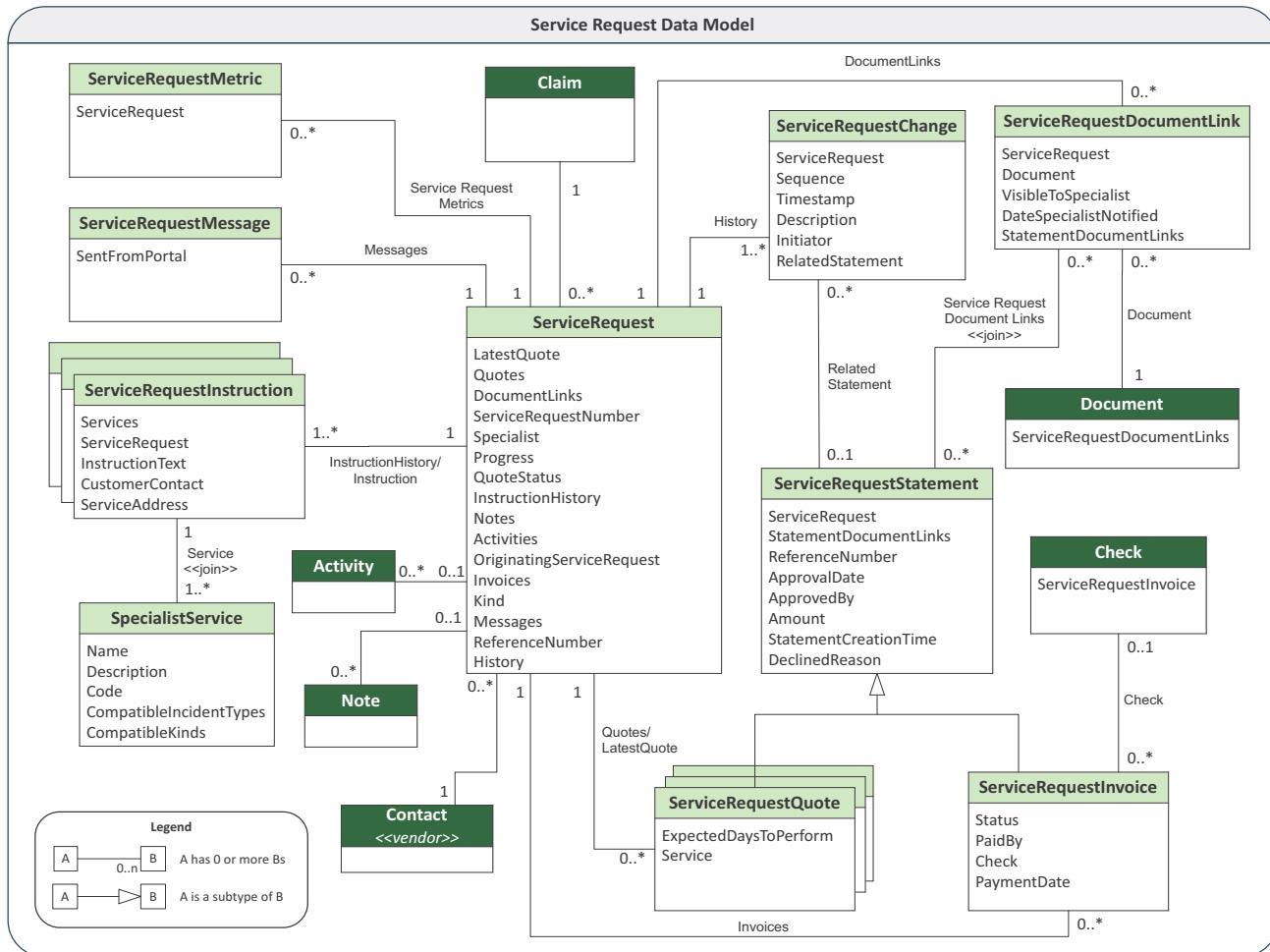
```
<Value>Kraftfahrt</Value>
</SpecialistService_Description_L10N>
</Description_L10N_ARRAY>
<Name>Auto</Name>
<Name_L10N_ARRAY>
  <SpecialistService_Name_L10N public-id="svc:1">
    <Language>es_PE</Language>
    <Owner public-id="svc:1" />
    <Value>Vehiculo</Value>
  </SpecialistService_Name_L10N>
<Name_L10N_ARRAY>
  <SpecialistService_Name_L10N public-id="svc:2">
    <Language>de_DE</Language>
    <Owner public-id="svc:1" />
    <Value>Kraftfahrt</Value>
  </SpecialistService_Name_L10N>
</Name_L10N_ARRAY>
</SpecialistService>
```

The `public-id` values have to be unique for each entity type, but not throughout the entire file. Three entities defined in the XML code are `SpecialistService`, `SpecialistService_Description_L10N`, and `SpecialistService_Name_L10N`, each of which has entities with their own sets of `public-id` values. Each L10N entity refers to the Auto `SpecialistService` entity with an `Owner` element set to that entity's `public-id`, `svc:1`.

Refer to the “Understanding String Resources” on page 36 in the *Globalization Guide* for more information on localizing services.

Service Request Data Model

The following object model diagram describes key entities relating to `ServiceRequest`. For complete information, see the Data Dictionary.



The following table describes the key entities.

Entity	Description
<code>ServiceRequest</code>	The core service request entity.
<code>ServiceRequestInstruction</code>	All instructions that have been created for this service request, including those that are no longer active. The <code>ServiceRequest</code> entity points to the latest instruction.
<code>ServiceRequestStatement</code>	An estimation, such as a quote or invoice, received from a vendor.
<code>ServiceRequestQuote</code>	Quotes received from a vendor for the service request. The <code>ServiceRequest</code> entity points to the latest version of the quote. <code>ServiceRequest</code> has an array of quotes only to keep a record of past quotes. <code>ServiceRequest.LatestQuote</code> is the only one considered to be in effect at a given time. This means the current quote is always expected to cover all the activity included in the service request.
<code>ServiceRequestInvoice</code>	Invoices associated with the service request.
<code>ServiceRequestDocumentLink</code>	Associates the service request to a document.
<code>ServiceRequestChange</code>	All changes that have been applied to this service request. These make up its history.

Entity	Description
SpecialistService	A node, signifying a service type, in the vendor service tree.
ServiceRequestMetric	Metrics related to the service request.
ServiceRequestMessage	Messages for the service request.

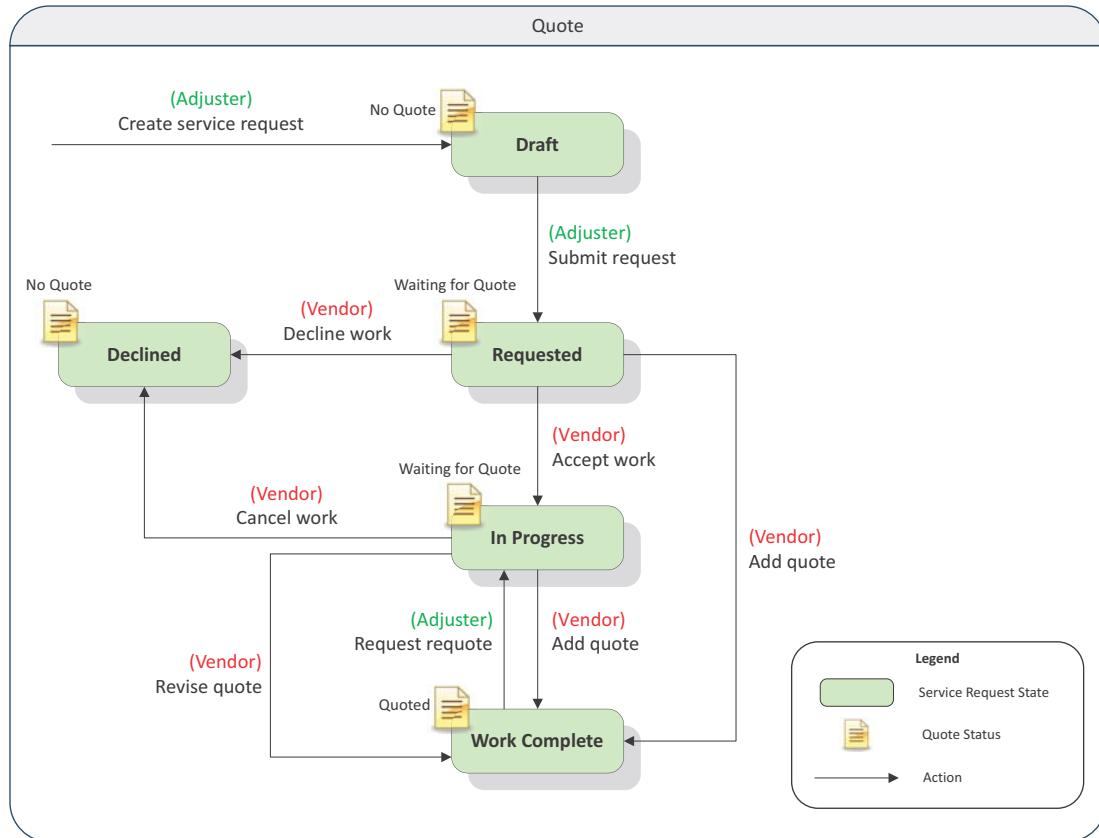
Lifecycle of a Service Request

Service requests can follow different paths in their progress to completion, based on the applicable *state handler*. State handlers are subclasses of `ServiceRequestStateHandler`, and they define the state transitions allowed for a service request. The state handler that applies to a `ServiceRequest` can be obtained by calling the method, `ServiceRequest.createStateHandler()`. In the base configuration, this method is implemented as a simple mapping between request types, modeled with `ServiceRequest.Kind`, and state handler implementation classes. Each state handler defines a set of operations and the availability and meanings of those operations. Because there is at most one active quote on a `ServiceRequest`, the quote status is stored on the `ServiceRequest` itself. The quote status, along with `ServiceRequest.Progress`, largely determine the state of a service request in its lifecycle.

The following state diagrams illustrate the lifecycle of the four basic service request types.

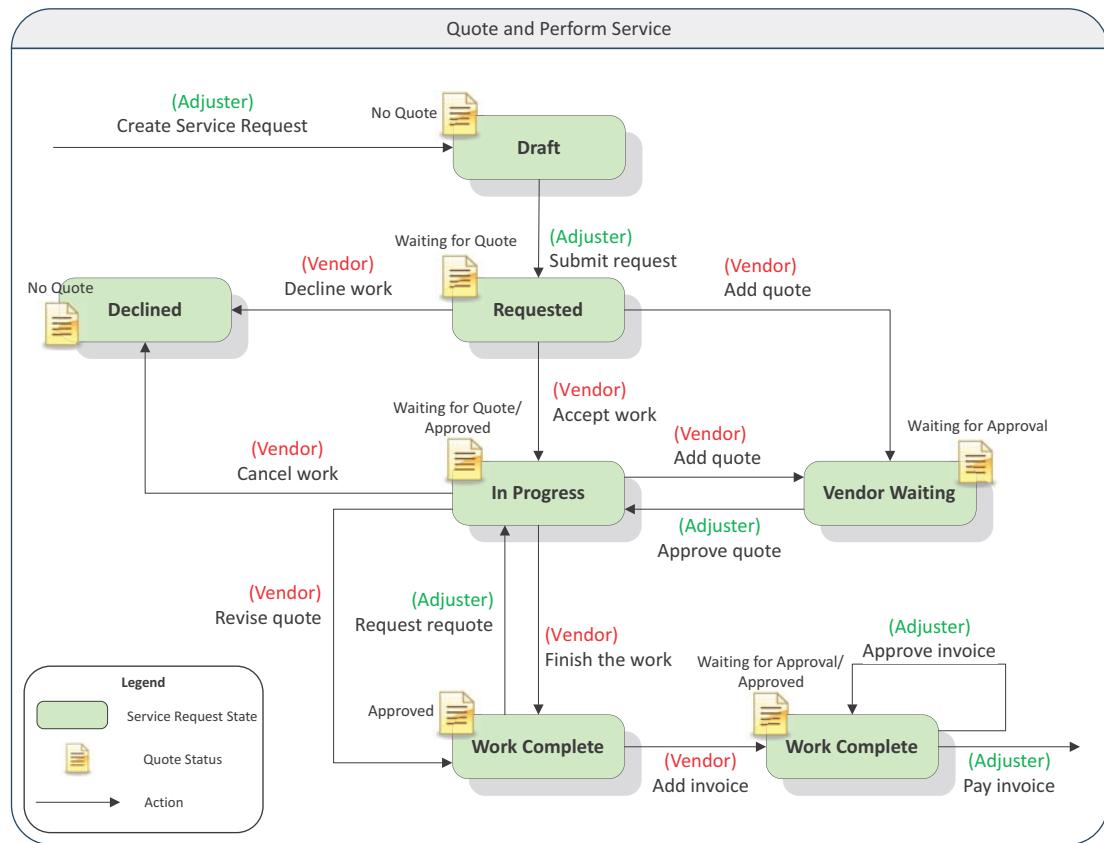
Quote Only

The following diagram displays the various stages in the lifecycle of a quote-only service request.



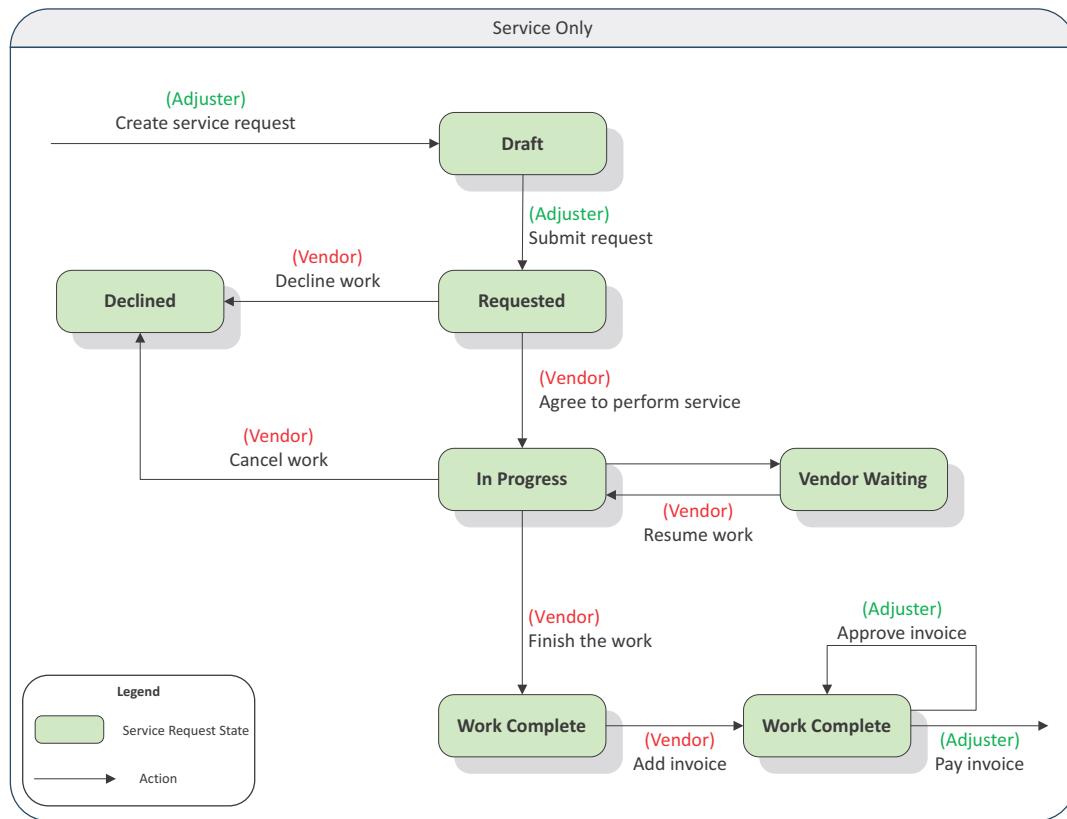
Quote and Perform Service

The following diagram displays the various stages in the lifecycle of a quote and perform service request.



Service Only

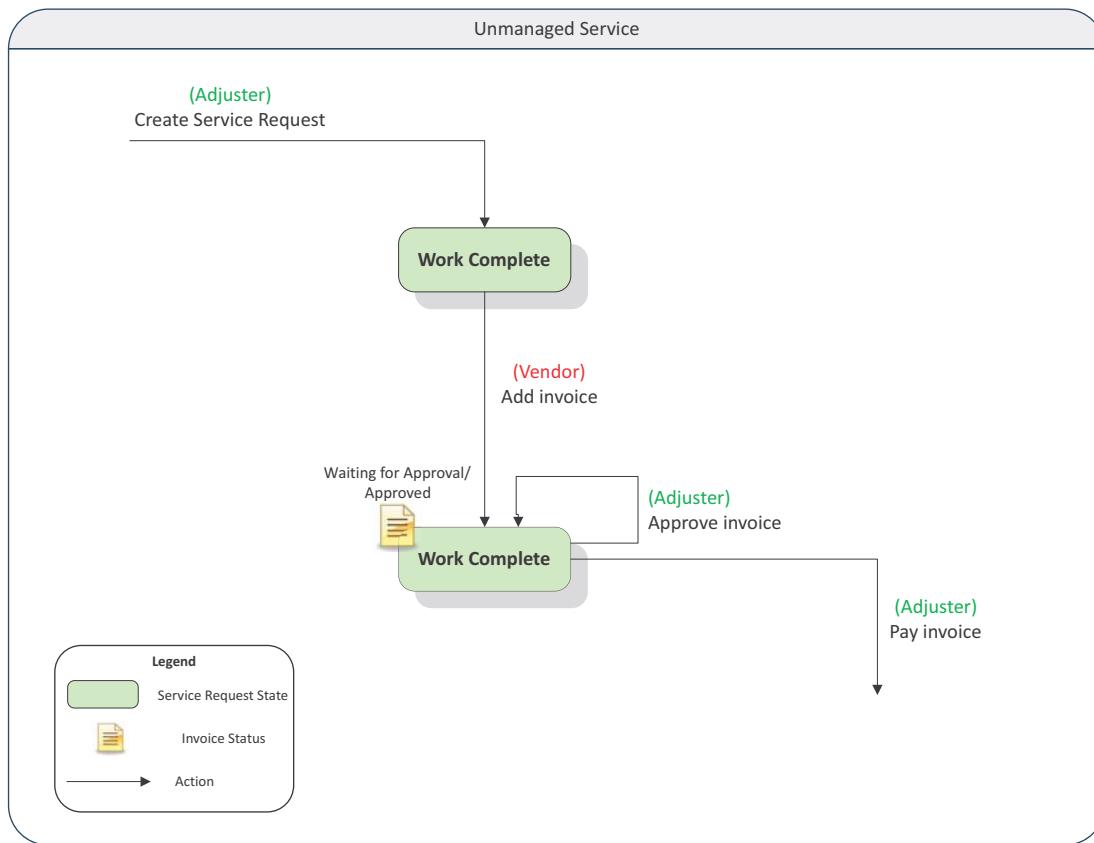
The following diagram displays the various stages in the lifecycle of a service-only request.



Unmanaged Service

Unmanaged services are specialized service request types created only as part of the Auto First and Final wizard. This service request type does not have associated quotes, and you can directly proceed to add invoices and make payments once the wizard is complete.

The following diagram illustrates the lifecycle of an unmanaged service request.



Configuring Service Metrics

ClaimCenter provides various measurements on the status and timeliness of service requests in the **Metrics** section of the **Services** screen.

In the base configuration, the following service request metrics are available:

- Response Time
- Quote Timeliness
- Service Timeliness
- Invoice Variance vs. Quote
- Number of Delays
- Cycle Time

Service request metrics are modeled after claim health metrics, and you can configure ClaimCenter to track additional, custom metrics.

See Also

- “Service Request Metrics” on page 386 in the *Application Guide*.
- “Configuring Claim Health Metrics” on page 553.

Example - Creating a Custom Service Request Metric

You can create a new service request metric by creating a subtype of an existing metric and adding implementation code. If you add a new subtype, the platform layer automatically adds a new member to the associated type-list. For example, if you add a new subtype of `MoneyServiceRequestMetric`, the system adds a new member to the `ServiceRequestMetric` typelist. Adding a new subtype is the only way of adding new members to this type-list.

In this example, a custom service request metric, `TestMoneyServiceRequestMetric`, is created to handle financial metrics related to service requests.

Step 1. Create a Subtype for the New Metric

The first step in creating a custom service request metric is to extend the `ServiceRequestMetric` entity. Typically, one of the supplied subtypes of this class, `IntegerServiceRequestMetric`, `DecimalServiceRequestMetric`, `MoneyServiceRequestMetric`, or `TimeBasedServiceRequestMetric`, can be extended, based on the metric type.

In this example, as a financial measurement is being tracked, the `MoneyServiceRequestMetric` entity is used to create a new subtype.

1. If necessary, start Guidewire Studio.

At a command prompt, navigate to the `ClaimCenter/bin` directory and enter:

```
gwcc studio
```

2. Navigate to **configuration** → **config** → **Extensions** → **Entity**.

3. Right-click and select **New** → **Entity**. Enter the following values:

Name	Value
Entity	<code>TestMoneySRMetric</code>
Entity Type	subtype
Desc	Custom money metric for service requests.
Supertype	<code>MoneyServiceRequestMetric</code>
Final	No

4. The `TestMoneySRMetric` entity editor is now shown. Click the plus icon (✚), and select **implementsInterface** from the drop-down choice list.

Enter the following values:

Name	Value
iface	<code>gw.api.metric.MetricMethods</code>
impl	<code>gw.vendormanagement.metric.TestMoneyServiceRequestMetricMethodsImpl</code>

The **Xml** tab now displays the following code:

```
<?xml version="1.0"?>
<subtype
  xmlns="http://guidewire.com/datamodel"
  desc="Custom money metric for service requests."
  entity="TestMoneySRMetric"
  supertype="MoneyServiceRequestMetric">
  <implementsInterface
    iface="gw.api.metric.MetricMethods"
    impl="gw.vendormanagement.metric.TestMoneyServiceRequestMetricMethodsImpl"/>
</subtype>
```

The new money metric subtype is automatically added to the `ServiceRequestMetric` typelist.

Note: The `TestMoneyServiceRequestMetricMethodsImpl` does not exist yet and will be created in the next step. You might see a corresponding error in Guidewire Studio.

Step 2. Add the New Metric to the Implementation Classes

1. Navigate to configuration → config → gsrc → gw → vendormanagement → metric, right-click and select New → Gosu Class.
2. Enter the following values:

Name	Value
Name	TestMoneyServiceRequestMetricMethodsImpl
Kind	Class

Click OK.

3. Edit the new class, `TestMoneyServiceRequestMetricMethodsImpl.gs`, that extends the `MoneyServiceRequestMetricMethodsImpl` class, as shown below.

```
package gw.vendormanagement.metric

uses gw.api.vendormanagement.metric.MoneyServiceRequestMetricMethodsImpl
uses gw.api.metric.MetricUpdateHelper
uses java.util.Date

class TestMoneyServiceRequestMetricMethodsImpl extends MoneyServiceRequestMetricMethodsImpl
{
    construct(moneyServiceRequestMetric : MoneyServiceRequestMetric)
    {
        super(moneyServiceRequestMetric)
    }

    override property get Metric() : MoneyServiceRequestMetric
    {
        return super.Metric
    }

    override function updateMetricValue(helper : MetricUpdateHelper) : Date
    {
        Metric.MoneyValue = 34.34
        return null
    }
}
```

4. Save your changes.

Step 3. Test the Changes

To test your changes:

1. Shut down ClaimCenter if it is running, and then restart it.
2. Log in to ClaimCenter as a user with the appropriate permissions to view and edit claims. For example, log in as user aapplegate with password gw.
3. Open an existing claim with service requests.
4. Select Services from the sidebar.
5. The new metric, `TestMoneyServiceRequestMetric`, is now visible in the Metrics section of the Services screen.

Configuring Service Request Metric Limits

Service request metrics are similar to claim health metrics with one difference – ClaimCenter provides the `IServiceRequestMetricLimitFinderPlugin` to allow for additional flexibility when configuring metric limits.

In the base configuration, `IServiceRequestMetricLimitFinderPlugin` retrieves service request metric limits, like other health metrics, from the ClaimCenter database. You can configure this plugin to compute limits based on claim and service request properties or to extract limit values from a contract system.

Unlike claim or exposure metrics, the `IServiceRequestMetricLimitFinderPlugin` extracts much of the functionality for finding and matching limits out into a separate class.

The plugin interface provides two methods:

- `findLimitValues` – Searches for the limit object matching the metric and service request passed in and performs calculations to return a set of target limit values. If there are multiple service requests, there may be multiple limits matched to the metric. In this case, the most lenient limit is returned.

In the base configuration, the following fields are used to match service requests to limits:

- `ServiceCategory`
- `SpecialistService`
- `ServiceRequestTier`
- `ServiceRequestMetric`
- `PolicyServiceTier`
- `limitValuesOutOfDate` – Examines potentially changeable fields on the limit object and returns true if the limit values that were matched before are now out of date. Examples of limit objects are `ServiceRequestTier`, `Currency`, or `PolicyServiceTier`.

See also

- “Claim Health Metrics” on page 392 in the *Application Guide*.

Configuring Deductibles

This topic explains how deductibles are structured, and how they interact with checks.

This topic includes:

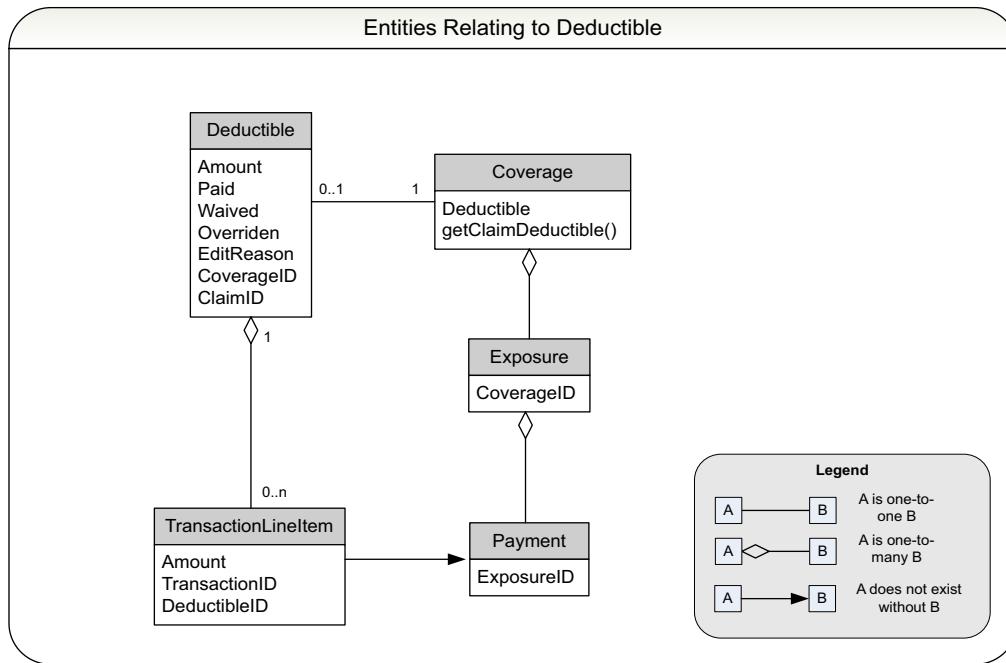
- “Deductible Data Model” on page 512
- “Typecodes for Deductibles” on page 513
- “Permissions for Deductibles” on page 513
- “Deductibles and Checks” on page 513
- “Deductibles and Rules” on page 515

See also

- “Deductible Handling” on page 349 in the *Application Guide* to learn about this feature.

Deductible Data Model

In this data model diagram, the main entity is **Deductible**. Every **Deductible** is associated with a **Coverage**, and by default gets its initial amount from the Coverage's deductible amount (the **Coverage.Deductible** field, which is of type **money**).



The **Deductible** entity has the following fields:

Field	Description
Amount	The amount that this deductible represents. This amount is specified in the claim currency.
Paid	Specifies whether this deductible has already been paid. This is initially false, and is set to true when a payment is created and a deductible applied to it.
Waived	Specifies whether this deductible has been waived. This is initially false, and can be set to true on the Exposure Edit page if the deductible has not been paid yet. If set to true, the check wizard does not allow this deductible to be applied to payments. In the database, however, if this has been set to true, this indicates that the amount has been modified.
Overridden	Specifies whether this deductible has been overridden and is initially set to false. If set to true, the amount field can be modified in the user interface.
EditReason	Specifies the reason why this deductible was waived or overridden.
CoverageID	Foreign key link to the coverage for which this deductible was calculated. This is nullable in the database to support policy-level deductibles, but the base configuration does not support this.
ClaimID	Foreign key link to the claim on which this deductible lives.

There is a one-to-at-most-one relationship from **Coverage** to **Deductible**, and a method on **Coverage** to access the **Deductible** pointing to it (if any). **TransactionLineItem** has a foreign key to **Deductible**. A deductible may be paid over any number of **TransactionLineItems**, though in the base configuration, ClaimCenter only supports paying it over one **TransactionLineItem**.

Typecodes for Deductibles

There are two typecodes that relate to Deductibles, both from the `LineCategory` typelist:

- **Deductible** – Indicates that a `TransactionLineItem` is a deductible line item. In other words, it is the `TransactionLineItem` to which a paid deductible is linked.
- **Former Deductible** – Indicates that a `TransactionLineItem` was originally a deductible line item, but is one no longer. Former deductibles can result from a recoded payment, a transferred or deleted check, or an onset payment whose deductible could not be applied to it.

Both typecodes are valid for any `Exposure`, `CostType`, or `CostCategory` because a deductible can be applied to a payment of any reserve line. However, they are not valid for any `Matter`. Wherever `LineCategory` is editable in the user interface, such as in Step 2 of the check wizard, the deductible typecodes are filtered out from the list of available options. This filtering prevents the user from selecting either typecode for normal, non-deductible line items. Anywhere in the user interface where `LineCategory` typecodes display, if a line item has a category of *Deductible* or *Former Deductible*, you cannot edit the line category and the amount.

Permissions for Deductibles

The permission `EditDeductible` allows you to edit, waive, or override the deductible on a claim file.

Deductibles and Checks

This section describes how deductibles affect the following types of checks:

- “Transferring Checks” on page 513
- “Recoding Payments” on page 514
- “Deleting and Voiding/Stopping Checks” on page 514
- “Denying or Resubmitting Checks” on page 514
- “Applying Deductibles on Multicurrency Checks” on page 514
- “Cleared or Issued Checks” on page 514
- “Cloning Checks” on page 514

Transferring Checks

When transferring a check, any deductible line items on the original and offset payments become *Former Deductible*, and linked deductibles are unlinked. This is done through the method `check.unlinkDeductibles`, after the target check has been created, but before calling the `financials.CheckUtil.transferCheck` method, in the `doTransfer` method in `CheckTransfer.pcf` file.

For the onset payment, if the target exposure has a valid deductible, then the target exposure's deductible is applied to the onset payment. If it has not been paid, waived, and the amount and the claim currency are equal, then the deductible amount can be applied to the payment. Otherwise, no deductible are applied, and the onset payment's deductible line item instead becomes *Former Deductible*. This is accomplished by the method `check.linkDeductibles` before calling `financials.CheckUtil.transferCheck` in the `doTransfer` method in `CheckTransfer.pcf` file.

If a check is transferred to a claim whose claim currency is different from the original claim's currency, then the deductible is not applied on the target claim. This is because there is little meaning in comparing two deductibles in different currencies to see if they have the same amount.

Recoding Payments

When recoding a payment, any deductible line items on the original and offset payments become *Former Deductible*, and linked deductibles are unlinked. This is accomplished by calling `payment.unlinkDeductible` on the original payment after the onset payment has been created, but before calling `financials.FinancialsUtil.recodePayment`, in the `doRecode` method in `RecodePayment.pcf` file.

For the onset payment, if the target exposure has a valid deductible, the target exposure's deductible is applied to the onset payment. See “Transferring Checks” on page 513. Otherwise, no deductible is applied, and the onset payment's deductible line item instead become *Former Deductible*. This is accomplished by the method `payment.linkDeductible` before calling `financials.FinancialsUtil.recodePayment` in the `doRecode` method in `RecodePayment.pcf` file.

Deleting and Voiding/Stopping Checks

Deleting or voiding/stopping a check converts all its deductible line items to have the line category *Former Deductible*. This is not exposed in the user interface in a deleted check. The linked deductibles are unlinked and unpaid. This is accomplished by calling the `check.unlinkDeductibles` method before calling the methods: `check.delete` or `check.void` or `check.stop`.

Denying or Resubmitting Checks

Denying a check converts all its deductible line items to have the line category *Former Deductible*, and the linked deductibles are unlinked. If you resubmit this check, ClaimCenter attempts to relink all the deductible line items to their prospective deductibles, provided they are still valid. During relinking, ClaimCenter does not waive or pay the deductible line items. And, the relinked deductible line item have the same amount as the corresponding line item. Line items whose prospective deductibles are no longer valid remain unlinked.

Applying Deductibles on Multicurrency Checks

Deductible amounts are specified in the claim currency, that is the claim on which the deductible's coverage exists. You can apply a deductible in the check wizard. If you do, then the deductible line item that is being added has an amount whose claim amount is fixed to be the negative amount of the deductible amount. Contrary to normal (non-deductible) line items, if the currency or exchange rate on the check is changed, the deductible line item's transaction amount is recalculated. The recalculation is based on the new exchange rate. However, its claim amount remains fixed.

Usually the claim amount of a transaction line item always match the amount to its linked deductible. However, this is not the case when a foreign exchange adjustment is applied to a payment. In this instance, the deductible line item's claim amount can deviate slightly from its deductible amount due to rounding errors.

Cleared or Issued Checks

Clearing or issuing a check does not have any impact on the deductible or former deductible line items.

Cloning Checks

Cloning checks does not affect deductibles, as it does not copy the deductible or former deductible line items. You can see this is in the `CloneCheckWizard.pcf` file by calling the `check.removeClonedDeductibleLineItems` method on the new check when the first step is first entered. ClaimCenter alerts you when this occurs.

Deductibles and Rules

Rules determine when claim deductibles are created. They create the deductible entity, if it has not yet been created. They also check if the exposure's coverage is updated. The configurable rules are in the Pre-Update rule set category:

Pre-update rule set	Rule
Exposure Pre-update	<ul style="list-style-type: none"><i>Update Deductible On Updated Exposure Coverage</i><i>Update Deductible On Updated Coverage Deductible</i>
Transaction Pre-update	<ul style="list-style-type: none"><i>Unlink Deductible After Check Denial</i>

See “Preupdate Rule Set Category” on page 63 in the *Rules Guide* for more information on the Pre-update rules.



chapter 38

Configuring Weighted Workload Assignment

The weighted workload feature in ClaimCenter 8.0 enables the efficient balancing of assignable objects, such as claims or exposures, across users and user groups. Weighted workload balancing happens, for the most part, in the background and uses configurable elements such as conditions and calculated weights to estimate the amount of effort required.

A workload-aware assignable has the appropriate infrastructure and configuration to interact with weighted workload assignment.

This topic describes the configuration of weighted workload balancing and includes:

- “Enabling Weighted Workload” on page 517
- “Weighted Workload Permissions” on page 518
- “Weighted Workload Configuration” on page 521
- “Weighted Workload Data Model” on page 520

See also

- “Weighted Workload” on page 207 in the *Application Guide*.

Enabling Weighted Workload

Weighted workload assignment includes two configuration parameters in `config.xml` – `WeightedAssignmentEnabled` and `WeightedAssignmentGlobalDefaultWeight`. In the base configuration, weighted workload balancing is not enabled. The configuration parameter, `WeightedAssignmentEnabled`, enables or disables weighted workload assignment in ClaimCenter.

Use the steps below to enable weighted workload assignment:

1. Start Guidewire Studio and open `config.xml`.
2. Set the value of `WeightedAssignmentEnabled` to `true`.
3. Save your changes.

Setting `WeightedAssignmentEnabled` to `false` disables automatic calculation and updating of the weights of all assignable objects and the calculation of total workloads for users.

The second configuration parameter for weighted workload, `WeightedAssignmentGlobalDefaultWeight`, defines the default weight for all workload-enabled, assignable objects. In the base configuration, this value is set to 10 and applies to all open claims and exposures that do not match any existing workload classifications. This value must be a non-negative integer. You can also alter the default weight of specific sets of assignable objects. See “Configuring the Default Weight in Code” on page 524.

IMPORTANT Default workload weight changes have system-wide impact and must be made infrequently. If you change this value, you need to run the `UserWorkloadUpdate` worker queue to recalculate stored entity workload weights. See “User Workload Update” on page 143 in the *System Administration Guide*.

Weighted Workload Permissions

Distinct permissions exist for both viewing and editing weighted workload information.

These are as follows:

Permission Code	Description
<code>wwlview</code>	View workload classifications.
<code>wwlmanage</code>	Modify workload classifications.
<code>wwlview</code>	View supplemental weights on weighted workload entities.
<code>wwlmanage</code>	Modify supplemental weights on weighted workload entities.

In the base configuration, adjusters and claim supervisors do not have these permissions. The Super User has all permissions, and the Manager role is given permissions to view and edit supplemental weights only.

Workload Weight Recalculation

The workload weight of an assignable object is calculated based on the state of the object and its relationship with predefined, active workload classifications. See “Calculating Weights” on page 210 in the *Application Guide*.

Assignable weights are recalculated only when significant changes occur that might affect the workload. These events also trigger user workload recalculation.

These events include:

- Creation of an assignable object, such as new claim creation through the FNOL wizard.
- Reassignment.
- State changes, such as the closing or reopening of a claim.
- Modification of attributes that affect workload.

When one of these events occurs, it triggers immediate recalculation of the assignable object's weight. This weight is then stored on the assignable's `WorkloadWeight` attribute.

The following section details the attribute changes that trigger recalculation.

Claim

The following fields, when modified, trigger workload recalculation on `Claim` objects:

- `LOBCode`
- `LossCause`
- `Segment`

- `SupplementalWorkloadWeight`

Adding or deleting exposures from a claim triggers recalculation of workload weights. In addition, the following fields, when modified on associated exposures of a `Claim`, trigger recalculation on that claim:

- `PrimaryCoverage`
- `CoverageSubtype`
- `LossParty`

The following fields, when modified on the associated policy of a `Claim`, also trigger recalculation on that claim:

- `PolicyType`
- `CustomerServiceTier`

Exposure

The following fields, when modified, trigger workload recalculation on `Exposure` objects:

- `Segment`
- `PrimaryCoverage`
- `CoverageSubtype`
- `LossParty`
- `SupplementalWorkloadWeight`

The following fields, when modified on the parent claim of an exposure, triggers recalculation on that `Exposure`:

- `LOBCode`
- `LossCause`

The following fields, when modified on the associated policy of the parent claim of an `Exposure`, triggers recalculation on that `Exposure`:

- `PolicyType`

The following fields, when modified on an associated incident of an exposure, triggers recalculation on that `Exposure`:

- `Severity`

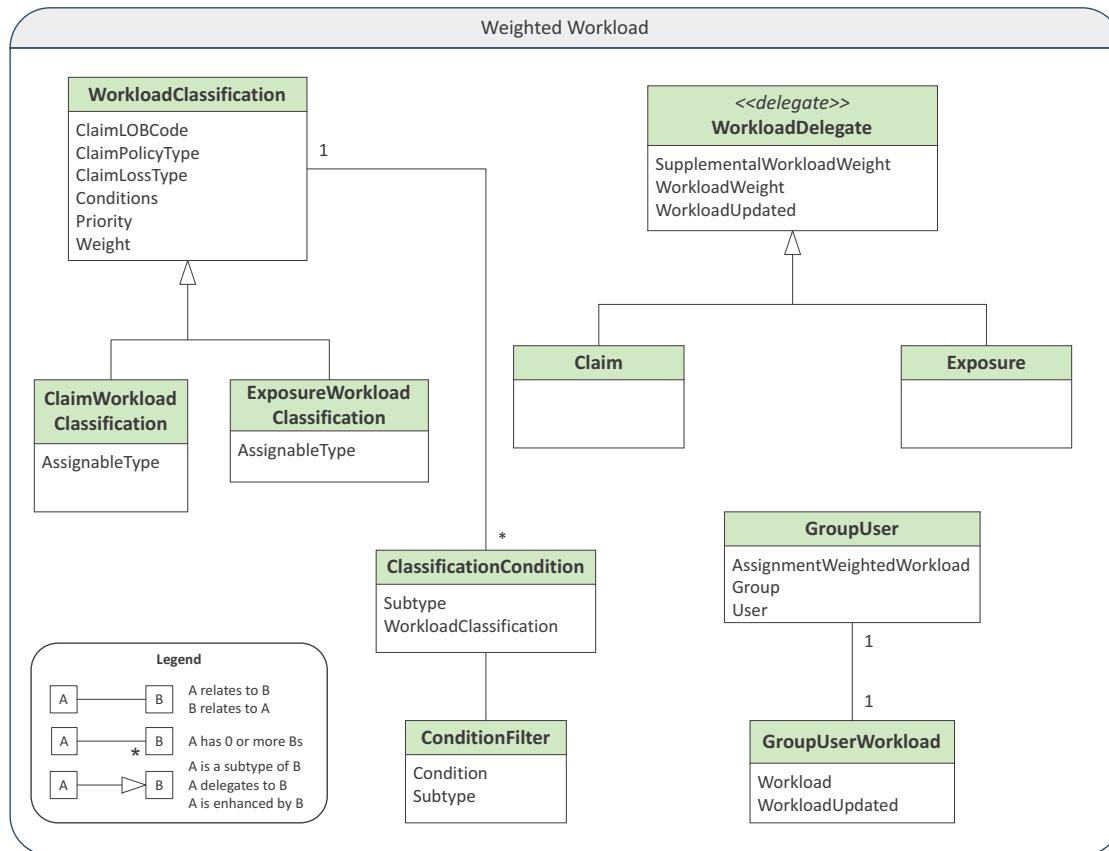
Closing Claims and Exposures

When a claim or an exposure is closed, the last calculated workload weight value for the claim or exposure is retained. The user's adjusted weight is then recalculated to exclude the closed claim or exposure.

If the claim or exposure is reopened subsequently, the weighted workload value is recalculated. See “Workload Weight Computation” on page 521.

Weighted Workload Data Model

The following object model diagram describes key entities relating to weighted workload assignment. For complete information, see the Data Dictionary.



The following table describes the key entities.

Entity	Description
WorkloadClassification	Entity that defines the workload classification, which defines the administrative processing of workload weights.
WorkloadDelegate	Delegate containing common fields used by workload-aware entities, including Claim and Exposure. These fields track the primary and supplemental workload weights and updates to the workload.
ClaimWorkloadClassification	Claim workload classification.
ExposureWorkloadClassification	Exposure workload classification.
ClassificationCondition	Filtering condition for workload classifications.
ConditionFilter	Type of filter associated with the workload classification condition. Subtypes include CustomerServiceTierConditionFilter, ExposureConditionFilter, IncidentSeverityConditionFilter, JurisdictionConditionFilter, LossCauseConditionFilter, and SegmentConditionFilter.
GroupUser	Links users to groups.
GroupUserWorkload	Tracks the amount of workload assigned to a group user.

Weighted Workload Configuration

Weighted workload assignment is not enabled in the base configuration, and you will need to enable it in config.xml. See “Enabling Weighted Workload” on page 517.

In addition to the configuration parameters, weighted workload configuration is comprised of the following two components:

- Weight Computation – If weighted workload is enabled in configuration, corresponding Preupdate rules are triggered for the assignable entity to compute the workload weight.
- User Assignment API – If weighted workload parameters and default assignment rules are enabled in configuration, user assignment is implemented using the default workload assignment strategy.

These components are described in further detail in this topic.

Workload Weight Computation

The first step in using weighted workload assignment is to enable this feature in configuration. See “Enabling Weighted Workload” on page 517.

Once weighted workload is enabled, associated Preupdate rules for **Workload Assignment Balancing** are activated. Any time relevant changes occur to workload-aware assignable objects, these rules are triggered to calculate the workload weight. Some events that activate these rules include the creation of a new claim or exposure, reassigning of a claim or exposure, and so on.

In the base configuration, the following rules are included:

ClaimPreupdate Rules

Rule	Description
CPU30000 – Workload Assignment Balancing	This parent rule and its child rules are activated only if weighted workload assignment is enabled in configuration. Checks gw.api.system.CCConfigParameters.WeightedAssignmentEnabled.Value.
CPU30100 – Claim Closed	Updates the weighted workload value when a claim is closed.
CPU30200 – Claim Reassignment	Updates the weighted workload value when the assigned user on a claim changes.
CPU30300 – Claim Reopened	Updates the weighted workload value when a claim status changes from “closed” to “open.”
CPU30400 – New Claim	Calculates the weighted workload value on a new claim.
CPU30500 – Claim Workload Affected	Updates the weighted workload value when a claim field that impacts workload is altered. See “Workload Weight Recalculation” on page 518.

ExposurePreupdate Rules

Use these rules to manage weighted workload assignment when specific changes take place in the assignable object, in this case, the exposure.

Rule	Description
EPU10000 – Workload Assignment Balancing	This parent rule and its child rules are activated only if weighted workload assignment is enabled in configuration. Checks <code>gw.api.system.CCConfigParameters.WeightedAssignmentEnabled.Value</code> .
EPU10100 – Exposure Closed	Updates the weighted workload value when an exposure is closed.
EPU10200 – Exposure Reassignment	Updates the weighted workload value when the assigned user on an exposure changes.
EPU10300 – Exposure Reopened	Updates the weighted workload value when the status of an exposure changes from “closed” to “open.”
EPU10400 – New Exposure	Calculates the weighted workload value on a new exposure.
EPU10500 – Exposure Workload Affected	Updates the weighted workload value when a field on the exposure that impacts workload is altered. See “Workload Weight Recalculation” on page 518.

Global Workload Assignment Rules

In order to use weighted workload assignment, you need to implement global assignment rules that utilize the Weighted Workload API commands.

In the base configuration, a set of sample global assignment rules using workload-aware assignment methods are provided. By default, these rules are disabled. Enable these rules or create your own to activate weighted workload assignment.

User Assignment API

In the base configuration, sample rules are provided for user assignment within a group. The user assignment API uses the default workload assignment strategy. You can customize both the methods and the assignment strategies used, as needed. See “Weighted Workload Assignment Strategies” on page 523.

Note: The sample rules for user assignment using weighted workload are disabled in the base configuration. You must activate them in order to start assigning claims and exposures based on workloads.

The following default rules are available.

Default Group Claim Assignment Rules

Rule Set	Rule	Description
DefaultGroupClaimAssignmentRules	DGC00500 – Balanced workload within group	This rule is activated only if weighted workload assignment is enabled in configuration. Assigns the claim to the most suitable user, that is, the user with the appropriate status, membership, and the lowest workload. Uses the default assignment strategy, <code>GroupUserWorkloadAssignmentStrategy</code> .

Default Group Exposure Assignment Rules

Rule Set	Rule	Description
DefaultGroupExposureAssignmentRules	DGC00500 – Balanced workload within group	<p>This rule is activated only if weighted workload assignment is enabled in configuration.</p> <p>Assigns the exposure to the most suitable user, that is, the user with the appropriate status, membership, and the lowest workload. Uses the default assignment strategy, <code>GroupUserWorkloadAssignmentStrategy</code>.</p>

Weighted Workload Assignment Strategies

Weighted workload assignment defines and uses dynamic assignment strategies to process and make decisions on how workload-enabled assignable objects must be assigned. Assignment strategies include support for workload-based and load factor-based assignment. Assignment strategies are located in `gw.assignment.workload.strategies`.

In the base configuration, ClaimCenter contains four default assignment strategies for specific use with weighted workload:

- `GroupUserWorkloadAssignmentStrategy`
- `GroupUserByAttributeWorkloadAssignmentStrategy`
- `UserWorkloadAssignmentStrategy`
- `UserByAttributeWorkloadAssignmentStrategy`

These four assignment strategies provide basic weighted workload assignment support, and you can choose to use or modify to fit your own requirements. Alternately, you can use these as a reference if you decide to implement your own strategies. These strategies fully support subgroup recursion.

GroupUserWorkloadAssignmentStrategy (Default)

`GroupUserWorkloadAssignmentStrategy` chooses from among a set of candidate `GroupUser` objects the group user who has the lowest workload weight and is the winner of ties, if any. In the base configuration, this is the default strategy used by the sample weighted workload assignment methods provided.

GroupUserByAttributeWorkloadAssignmentStrategy

`GroupUserByAttributeWorkloadAssignmentStrategy` is functionally identical to `GroupUserWorkloadAssignmentStrategy`, but allows the caller to specify user attribute criteria to filter down the list of candidate users before the decision-making process begins.

UserWorkloadAssignmentStrategy

`UserWorkloadAssignmentStrategy` chooses from among a set of candidates the group user who has the least weight and who is the winner of ties, if any. This selection is based on the absolute weight of the group user rather than the group weight. See “Calculating Weights” on page 210 in the *Application Guide*.

Unlike `GroupUserWorkloadAssignmentStrategy`, the user's calculated weight across the entire system is taken into account rather than just within a particular group.

UserByAttributeWorkloadAssignmentStrategy

`UserByAttributeWorkloadAssignmentStrategy` is functionally identical to `UserWorkloadAssignmentStrategy`, but allows the caller to specify user attribute criteria to filter down the list of candidate users before the decision-making process begins.

Custom Configuration

You can customize multiple components in the weighted workload assignment API. This topic includes examples on configuring the default weight, weighted workload strategies, and classifications.

Configuring the Default Weight in Code

The default weight is the workload weight assigned to assignable objects that do not match any existing weighted workload classification. This default value can be defined globally in the configuration parameter, `WeightedAssignmentGlobalDefaultWeight`, or in configuration code, as described in this topic. See “Enabling Weighted Workload” on page 517 for details on setting the global default weight.

You can specify the default weight for a class of workload weight-aware assignable objects. First, you need to change the `WorkloadProxy` descendant class for the assignable class to override the global default weight property.

For example, to set the default weight of all claims to 15, modify the `gw.assignment.workload.proxy.ClaimWorkloadProxy` class, as follows:

```
public class ClaimWorkloadProxy extends AbstractWorkloadProxy {  
    ...  
    final public override property get DefaultWeight() : int {  
        return 15  
    }  
    ...  
}
```

The global default weight property defined in `config.xml` is retrieved and stored in the `AbstractWorkloadProxy` class. Now, this value is overridden and the default workload weight for all claims without any matching workload classification is set to 15.

Note: The value for the default workload weight must be a constant, non-negative integer.

Creating Custom Workload Strategies

The determination of candidate assignees by weighted workload assignment is made by dynamic assignment strategies that leverage the computed weights of assignable entities. In the base configuration, four basic assignment strategies are included as examples. See “Weighted Workload Assignment Strategies” on page 523.

You can create your custom weighted workload assignment strategy. It is recommended that your custom assignment strategy class extend the `gw.api.assignment.workload.strategies`.

`AbstractWorkloadAssignmentStrategy` class. This is the base class specifically designed for workload strategy customization.

Example - Creating a Custom `WorkloadAssignmentStrategy`

Consider a ClaimCenter user who is a member of several groups. You can create a custom weighted workload assignment strategy that uses both the user’s group-level and system-level workload values.

Use the following steps to create the custom strategy:

1. Create a custom assignment strategy class.

```
public class CustomWorkloadAssignmentStrategy extends AbstractWorkloadAssignmentStrategy {  
    ...  
    protected override function fetchWorkload(groupUser : GroupUser) : int {  
        // The weighted workload of the user is now user's workload for the group they are  
        // currently in + half their overall total system workload  
        return groupUser.AssignmentWeightedWorkload + (groupUser.User.TotalWorkload / 2)  
    }  
}
```

2. Override the workload assignment method.

Now that the custom weighted workload strategy has been created, the system needs to be set up to use it. You can do this in two different ways:

- Override the assignment method in the corresponding `WorkloadDelegate` entity. For example, override the `assignUserByWorkload()` in `ClaimWeightedWorkloadMethodsImpl`, as follows:

```
public override function assignUserByWorkload(includeSubgroups:boolean, withinGroup:Group): boolean
{
    var result : boolean
    result = Owner.assignUserDynamically(new CustomWorkloadAssignmentStrategy(), withinGroup,
        includeSubgroups)
    return result
}
```

- Alternately, you can incorporate the new assignment strategy in a default group assignment rule for the assignable entity. See “Workload Weight Computation” on page 521. In the following example, the default group claim assignment rule for weighted workload, DGC00500, is modified:

```
...
static function doAction(claim : entity.Claim, actions : gw.rules.Action) {
    /*start00rule*/
    ...
    var assignmentSuccess = claim.assignUserDynamically(new CustomWorkloadAssignmentStrategy(),
        withinGroup, includeSubgroups)

    ...
}
```

Adding Criteria to Workload Classifications

Weighted workload classifications specify criteria used to classify incoming assignable objects during assignment. You can add and delete workload classifications using the Administration menu. See “Weighted Workload Classifications” on page 209 in the *Application Guide*.

You can also customize workload classifications by adding additional criteria, which can be set up to either match or not match the assignable object’s attributes.

In the base configuration, the following basic criteria are included in workload classifications:

- Claim Loss Type
- Claim Line of Business
- Claim Policy Type

You can add your own additional criteria, as illustrated in the next example, using the following steps:

- “Step 1. Add the New Classification Criterion” on page 526
- “Step 2. Add the New Criterion to the Implementation Classes” on page 526
- “Step 3. Modify the ClaimCenter User Interface” on page 527
- “Step 4. Test the Changes” on page 527

In the base implementation, simple criteria mostly look at intrinsic attributes on the Assignable entity itself using only equivalence; however, criteria are not restricted to these. Attributes for criteria can also be on subentities pointed to by foreign keys, and criteria can include equivalence checks, range checks, or any other SQL expression for an attribute. Note that the complexity of the criteria will affect scalability and performance.

Example - Add Flag Status to Claim Workload Classification

In the following example, the claim’s flag status is added as a weighted workload classification criterion.

Step 1. Add the New Classification Criterion

Add the new criterion, `ClaimFlagged`, to the `ClaimWorkloadClassification` entity, as shown below.

1. If necessary, start Guidewire Studio.

At a command prompt, navigate to the `ClaimCenter/bin` directory and enter:

```
gwcc studio
```

2. Navigate to **configuration** → **config** → **Extensions** → **Entity** and open `ClaimWorkloadClassification.eti`.
3. Click the plus icon (✚), and select **typekey** from the drop-down choice list.
4. Enter the following values:

Name	Value
<code>name</code>	<code>ClaimFlagged</code>
<code>typelist</code>	<code>FlaggedType</code>
<code>nullok</code>	<code>true</code>
<code>desc</code>	<code>Flag status of the Claim.</code>

Step 2. Add the New Criterion to the Implementation Classes

1. Add checks for the flag attribute in `ClaimWorkloadClassificationMethodImpl`, which is the implementation of the `WorkloadClassificationMethods` delegate for the `Claim` entity.

In Studio, navigate to **configuration** → **config** → **gsrc** → **gw** → **assignment** → **workload** → **classifications** and open `ClaimWorkloadClassificationMethodsImpl`. Modify the `isMatch()` and `buildQuery()` methods to include the new flag status criterion, as follows.

```
public override function isMatch(entity: Bean): boolean
{
    ...
    // Check if the claim's Flagged setting matches the predefined workload classification.
    if (claim.Flagged != (WorkloadClassification as ClaimWorkloadClassification).ClaimFlagged)
    {
        return false
    }
    ...
}

protected override function buildQuery(query: Query)
{
    ...
    // Claim Flagged matches
    query.compare(Claim#Flagged, Equals, (WorkloadClassification as ClaimWorkloadClassification).
        ClaimFlagged)
    ...
}
```

2. The next step is to override `ClaimWeightedWorkloadMethodsImpl`, which is the implementation of `WeightedWorkloadMethods` for the `Claim` entity.

In Studio, navigate to **configuration** → **config** → **gsrc** → **gw** → **assignment** → **workload** → **entity** and open `ClaimWeightedWorkloadMethodsImpl`. Add the new classification criterion to the `isWorkloadAffected()` method, which is used to check if the stored workload needs to be recalculated.

```
public class ClaimWeightedWorkloadMethodsImpl extends AbstractWeightedWorkloadMethodsBaseImpl<Claim>
{
    ...
    public override function isWorkloadAffected(): boolean
    {
        ...
        // Add new field to check if the Claim Flagged field has been changed
        if (Owner.isFieldChanged(Claim#Flagged))
        {
            return true
        }
        ...
    }
}
```

```
    }
```

Step 3. Modify the ClaimCenter User Interface

Modify the ClaimCenter user interface to add a field for the claim's flag status.

1. In Studio, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **admin** → **workload**, and open **WorkloadClassificationDV.ClaimWorkloadClassification.pcf**.
2. Double-click the **WorkloadClassificationCommonInputSet** to open it for editing.
3. Add a new **Input** widget below the **AllClaimPolicyType** widget, and enter the following properties (create new display keys, where required):

Name	Value
editable	false
id	AllClaimFlagStatus
label	displaykey.Web.Admin.Workload.WorkloadClassification.ClaimFlagStatus
required	true
value	(assignmentClassification as ClaimWorkloadClassification).ClaimFlagged

4. Add a new **Input** widget below the **ClaimPolicyType** widget, and enter the following properties:

Name	Value
editable	true
id	ClaimFlagStatus
label	displaykey.Web.Admin.Workload.WorkloadClassification.ClaimFlagStatus
required	true
value	(assignmentClassification as ClaimWorkloadClassification).ClaimFlagged

Step 4. Test the Changes

To test your changes:

1. Shut down ClaimCenter if it is running, and then restart it.
2. Log in to ClaimCenter as a user with the appropriate permissions to view and edit classifications. For example, log in as user su with password gw.
3. Navigate to **Administration** → **Business Settings** → **Weighted Workload**.
4. Select an active classification from the list. In the **Criteria** section, the **Claim Flag Status** field is now shown.
5. Click **Edit**. The **Claim Flag Status** field is now editable. Select a value from the drop-down list, and click **Update**.
6. ClaimCenter displays a message informing you that you need to run the batch process to update existing, open claims and exposures. Click **OK**.

The **Claim Flag Status** field now displays the value selected in Step 5.

Adding Workload Classification Conditions

ClaimCenter provides the ability to create or customize non-restrictive criteria, also called *conditions* in the data model, in classifications. Classification conditions can be restricted to specific condition filters. For example, the **Service Tiers** condition can be configured to filter by Gold customers only.

Assignable entities are only considered matches if they meet any of the condition filters defined on a classification.

IMPORTANT Custom conditions cannot be used with existing weighted workload classifications, if any. They can only be applied to new classifications. See “Editing Existing Classifications” on page 210 in the *Application Guide*.

In the base configuration, the following conditions are included:

- Exposures
- Claim Segments
- Claim Loss Causes
- Service Tiers

You can add your own custom condition, as illustrated in the next example, using the following steps:

- “Step 1. Add the New Classification Condition” on page 528
- “Step 2. Add the New Condition Filter” on page 529
- “Step 3. Add the New Condition to the Implementation Classes” on page 529
- “Step 4. Modify the ClaimCenter User Interface” on page 531
- “Step 5. Test the Changes” on page 532

Example - Add a Color Condition to Claim Workload Classification

In this example, a new weighted workload classification condition is created for the assignable entity, `Claim`. Assume that a custom typelist, `Color`, has been added to the `Claim` entity with predefined values such as `green`, `blue`, and `red`.

The goal is create a new condition for claim classifications based on the `color` field.

Step 1. Add the New Classification Condition

Create a new entity subtype of `ClassificationCondition` for the new condition. A weighted workload classification can have one and only one instance of any `ClassificationCondition`.

1. If necessary, start Guidewire Studio.

At a command prompt, navigate to the `ClaimCenter/bin` directory and enter:

```
gwcc studio
```

2. Navigate to **configuration** → **config** → **Extensions** → **Entity** and select **New** → **Entity**. Create `ColorCondition.eti`, as follows:

```
<?xml version="1.0"?>
<subtype xmlns="http://guidewire.com/datamodel"
  desc="Color Classification Condition"
  entity="ColorCondition"
  final="false"
  priority="1"
  supertype="ClassificationCondition">
  <implementsInterface
    iface="gw.api.assignment.workload.classifications.conditions.ConditionMethods"
    impl="gw.assignment.workload.classifications.conditions.ColorConditionMethodsImpl"/>
</subtype>
```

Note: The new subtype, `ColorCondition`, must implement the `ConditionMethods` interface in order to work with the weighted workload system.

3. Add the new condition subtype to the `ClassificationCondition` typelist, as follows:

1. Navigate to **configuration** → **config** → **Extensions** → **Typelist** and open `ClassificationCondition.ttx`.

2. Click the plus icon (⊕), and select **typecode** from the drop-down choice list.

3. Enter the following values:

Name	Value
code	ColorCondition
name	Color Classification Condition
desc	Classification condition filter by claim color
priority	1
retired	false

Step 2. Add the New Condition Filter

Create a new entity subtype of **ConditionFilter**, which defines the filters that restrict the condition, if selected. When a classification condition has a single condition filter, the condition is satisfied if the assignable's properties match the criteria of the condition filter. If a classification condition has multiple condition filters, the condition is considered satisfied if an assignable's properties match any of the condition filters.

1. Navigate to **configuration → config → Extensions → Entity** and select **New → Entity**. Create **ColorConditionFilter.eti**, as follows:

```
<subtype xmlns="http://guidewire.com/datamodel"
desc="Classification condition filter by Color"
entity="ColorConditionFilter"
final="false" priority="1"
superType="ConditionFilter">
<typekey
desc="Classification condition filter by Color"
name="Color"
nullOk="false"
typeList="Color"/>

<index name="clr_cond_index_1"
desc="Prevents duplicate condition filters"
unique="true">
<indexcol name="ClassificationConditionID" keyPosition="1"/>
<indexcol name="Color" keyPosition="2" />
</index>
</subtype>
```

2. Add an array of the new filter subtype, **ColorConditionFilter**, to the new condition subtype, **ColorCondition**, created in Step 1. The array's values map to the filtered values for this custom condition.

```
<array
arrayEntity="ColorConditionFilter"
name="ConditionFilters"
cascadeDelete="true"
/>
```

Step 3. Add the New Condition to the Implementation Classes

Create a new class, **ColorConditionMethodsImpl**, that extends the **AbstractConditionMethodsImpl** class, as shown below.

```
public class ColorConditionMethodsImpl extends AbstractConditionMethodsImpl
{
    // Because of delegation, this MUST always be present.
    public construct(filterSet : ColorCondition)
    {
        super(filterSet)
    }

    public override function filterQuery(query : Query)
    {
        var ColorCondition = (Condition as ColorCondition)

        // This classification condition only works with claim classifications.
    }
}
```

```
if (not (ColorCondition.WorkloadClassification typeis ClaimWorkloadClassification))
{
    return
}

// Filter by specific claim loss causes, if requested.
if (not Condition.IncludeAll and ColorCondition.ConditionFilters.HasElements)
{
    var Colors = ColorCondition.ConditionFilters.map(\ cause -> cause.Color)
    query.and(\ andExp ->
    {
        query.compareIn("Color", Colors)
    })
}

public override function isMatch(entity : Bean) : boolean
{
    switch(typeof entity)
    {
        // If this entity is a Claim, then check if the claim's color matches the color specified by the
        // condition filter.
        case Claim:
            return matchesColor(entity.Color)

        // This is not a Claim, do nothing.
        default:

            // Ignore as if it didn't exist.
            return true
    }
}

public override property get HasFilters() : boolean
{
    return not (Condition as ColorCondition).ConditionFilters.IsEmpty
}

public override function clearFilters()
{
    var cond = (Condition as ColorCondition)
    for (filter in cond.ConditionFilters)
    {
        cond.removeFromConditionFilters(filter)
    }
}

private function matchesColor(Color : Color) : boolean
{
    var result = false

    // If "Include All" is not selected, go through all the condition filters attached to this
    // classification condition. If any match, the condition is satisfied.
    if (not Condition.IncludeAll)
    {
        var filterSet = (Condition as ColorCondition)
        if (not filterSet.ConditionFilters.IsEmpty)
        {
            result = filterSet.ConditionFilters.hasMatch(\ lcf ->lcf.Color == Color)
        }
    }

    // If "Include All" is selected, there is always a match.
    else
    {
        result = true
    }
    return result
}
```

Step 4. Modify the ClaimCenter User Interface

Modify the ClaimCenter user interface to add a field for the claim's color status.

1. In Studio, navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **admin** → **workload**, and open **WorkloadClassificationDV.ClaimWorkloadClassification.pcf**.
2. Add a new **BooleanRadioInput** widget and enter the following properties (create new display keys, where required):

Name	Value
editable	true
falseLabel	displaykey.Web.Admin.Workload.WorkloadClassification.RestrictedTo
id	AllColors
label	displaykey.Web.Admin.Workload.WorkloadClassification.AllColors
required	true
trueLabel	displaykey.Web.Admin.Workload.WorkloadClassification.All
value	classification.ColorCondition.IncludeAll

3. Navigate to **configuration** → **config** → **Page Configuration** → **pcf** → **admin** → **workload** → **conditions**. Right-click and select **New PCF File**.
4. In the **File name** field, enter **ColorConditions**. In the **File type** menu, select **List View**.
5. Add the following properties for **ColorConditionsLV**:

Name	Value
Properties Tab	
id	ColorConditions
Required Variables Tab	
name	classification
type	WorkloadClassification
Exposes Tab	
valueType	ColorConditionFilter
widget	ColorConditionsLV

6. Add a **RowIterator** with the following properties:

Name	Value
editable	true
elementName	color
toAdd	classification.ColorCondition.addToConditionFilters(color)
toRemove	classification.ColorCondition.removeFromConditionFilters(color)
value	classification.ColorCondition.ConditionFilters
canPick	true
hideCheckboxesIfReadOnly	true

7. Add a Row widget. Add a RangeCell1 widget inside with the following properties:

Name	Value
editable	true
id	ColorConditionFilter
label	displaykey.Web.Admin.Workload.WorkloadClassification.ColorConditionFilter.Color
required	true
value	color.Color
valueRange	typekey.Color.getTypeKeys(false)

8. Open `WorkloadClassificationDV.ClaimWorkloadClassification.pcf`. Under the `AllColors` widget added in Step 2, add a `ListViewInput` widget and define the properties below.

Name	Value
def	ColorConditionsLV
id	ColorConditions
labelAbove	true
editable	not classification.ColorCondition.IncludeAll
visible	not classification.ColorCondition.IncludeAll

9. Add a `Toolbar` widget inside the `ColorConditions` widget created in Step 8. Add an `IteratorButtons` widget inside the toolbar with the following properties:

Name	Value
iterator	ColorConditions.ColorConditionsLV
showAddConfirmMessage	true
showRemoveConfirmMessage	true

10. Navigate to `configuration → config → Page Configuration → pcf → admin → workload` and open `NewWorkloadClassificationPopup.pcf`

11. Select the `Code` tab and add the new condition:

```
switch(entityType)
{
    case ClaimWorkloadClassification:
        ...
        result.addToConditions(new ColorCondition())
        ...
}
```

12. Save your changes.

Step 5. Test the Changes

To test your changes:

1. Shut down ClaimCenter if it is running, and then restart it.
2. Log in to ClaimCenter as a user with the appropriate permissions to view and edit classifications. For example, log in as user su with password gw.
3. Navigate to `Administration → Business Settings → Weighted Workload`.

4. Click **Add Classification** → **Add Claim Classification** to create a new classification. Enter **General** information and required criteria. See “Adding Classifications” on page 209 in the *Application Guide*.
5. In the **Criteria** section, the **Colors** field is now shown. Click **Restrict to any of the following:**, select **Add**, and choose a color from the drop-down menu.
6. Click **Update**, then **OK**.
7. Create a new claim with the requisite color selection. Verify that the claim is assigned the correct weight and classification.

Working with Catastrophe Bulk Associations

This topic explains how to configure Catastrophe Bulk Associations batch job, which is a Gosu batch process.

This topic includes:

- “Catastrophe Bulk Association Configuration” on page 535
- “Catastrophes Data Model” on page 536
- “Catastrophe Configuration Parameter” on page 536

See also

- “Catastrophes and Disasters” on page 155 in the *Application Guide* to learn about catastrophes in general
- “Catastrophe Bulk Association” on page 158 in the *Application Guide* to learn about Catastrophe Bulk Associations batch process
- “Managing Catastrophes” on page 475 in the *Application Guide* to learn how to administer catastrophes

Catastrophe Bulk Association Configuration

You can optionally configure the Catastrophe Bulk Associations batch job.

The two files you need are:

- `GWCatastropheEnhancement.gsx`
- `CatastropheClaimFinderBatch.gs`

You first need to define your new method in the `GWCatastropheEnhancement.gsx` file and second, point to it from the `CatastropheClaimFinderBatch.gs` file. The files are located in Studio.

Navigate to:

- `configuration` → `Classes` → `gw` → `util` → `CatastropheClaimFinderBatch`
- `configuration` → `Classes` → `gw` → `entity` → `GWCatastropheEnhancement`

The CatastropheClaimFinderBatch Job

`CatastropheClaimFinderBatch` (the batch job for catastrophe bulk association) is a subtype of `BatchProcessBase`. It finds the claims that have been defined by the `GWCatastropheEnhancement` class and creates a *review for catastrophe* activity (activity pattern code name `catastrophe_review`) if one does not already exist.

Note: In the base configuration, the batch process defines the areas by zones.

The GWCatastropheEnhancement Class

This configurable entity finds all claims that might be a match to the defined catastrophe.

It checks to see if the claim matches certain criteria. A match occurs if:

- The claim has not already been associated with a catastrophe
- The claim's loss date falls within the catastrophe's valid dates
- The catastrophe's perils list the claim's loss type and loss cause
- The claim does not have the `catastrophe_review` activity pattern with a *skipped* or *complete* status

The method `findClaimsByCatastropheZone` finds claims by zones. You can define the zone criteria. Examples might be defined as: United States states, regions (southern California, Northern California), territories (western territories such as California, Nevada, Oregon, and Washington).

Lastly, it returns all claims that match that criteria.

Catastrophes Data Model

The system uses the following entities and typelists to add catastrophes to the database:

Entity or Typelist	Description
<code>Catastrophe</code>	The main entity contains all the information for each catastrophe. It uses these two array
<code>CatastrophePeril</code> (virtual array)	entities to store each catastrophe's perils and the states in which it is valid.
<code>CatastropheZone</code> (virtual array)	
<code>CatastropheType</code> (typelist)	Whether the catastrophe came from ISO data (<code>iso</code>) or was manually entered (<code>internal</code>).

Catastrophe Configuration Parameter

The `MaxCatastropheClaimFinderSearchResults` parameter has a default of 1000. It limits the number of claims that can be found to match the criteria. For example, if there are 5000 claims that match, then the `CatastropheClaimFinder` batch process gets the claims that match the criteria. It creates a *Review for Catastrophe* activity for the first 1000. The next 1000 are processed during the next scheduled batch process and this process continues until there are no more claims that meet the criteria.

Configuring the Catastrophe Dashboard

This topic describes how to configure the Catastrophe Dashboard. This dashboard shows a map of policies and claims for a catastrophe area and provides various kinds of information on claims and policies related to the catastrophe. For information on the dashboard itself, including information on how to open it and use it, see “Catastrophe Dashboard” on page 160 in the *Application Guide*.

This topic includes:

- “Technical Design” on page 537
- “Configuring the Heat Map” on page 544

See also

- “Catastrophes and Disasters” on page 155 in the *Application Guide* to learn about catastrophes in general
- “Enabling Catastrophe Search and Heat Maps” on page 18 in the *System Administration Guide* to get the Catastrophe Dashboard, including geocoding, up and running

Technical Design

This topic describes the technical features, including classes, PCF files, and web services, used to generate the Catastrophe Dashboard.

This topic includes:

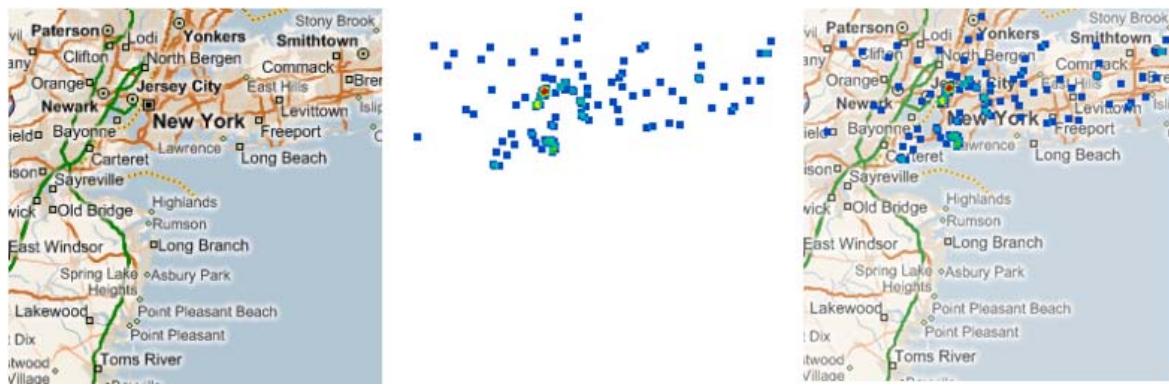
- “Heat Map Generation and Components” on page 538
- “Server, Browser, and Service Interactions” on page 539
- “Principal Heat Map Classes and Files” on page 540
- “Datasets and Map Data Points” on page 541
- “Datasets and Caching” on page 541
- “Datasets and Filtering” on page 542

- “DBA Scripts to Improve Oracle and SQL Server Performance” on page 542

Heat Map Generation and Components

The browser generates the heat map graphic by combining map images from a map service with an overlay image generated by the Guidewire server. The overlay image shows the map data points in color. The base configuration can use the Bing Maps service. Map images and the overlay are organized into 256 x 256 pixel tiles.

Following is an example showing, left to right, the map tile, the overlay tile, and the combined image. By default, the overlay image is opaque where there is color, with a semitransparent background. The default background color is white and 30% opaque to make the map colors slightly washed out so the overlay colors stand out better.



The components used to generate the heat map are:

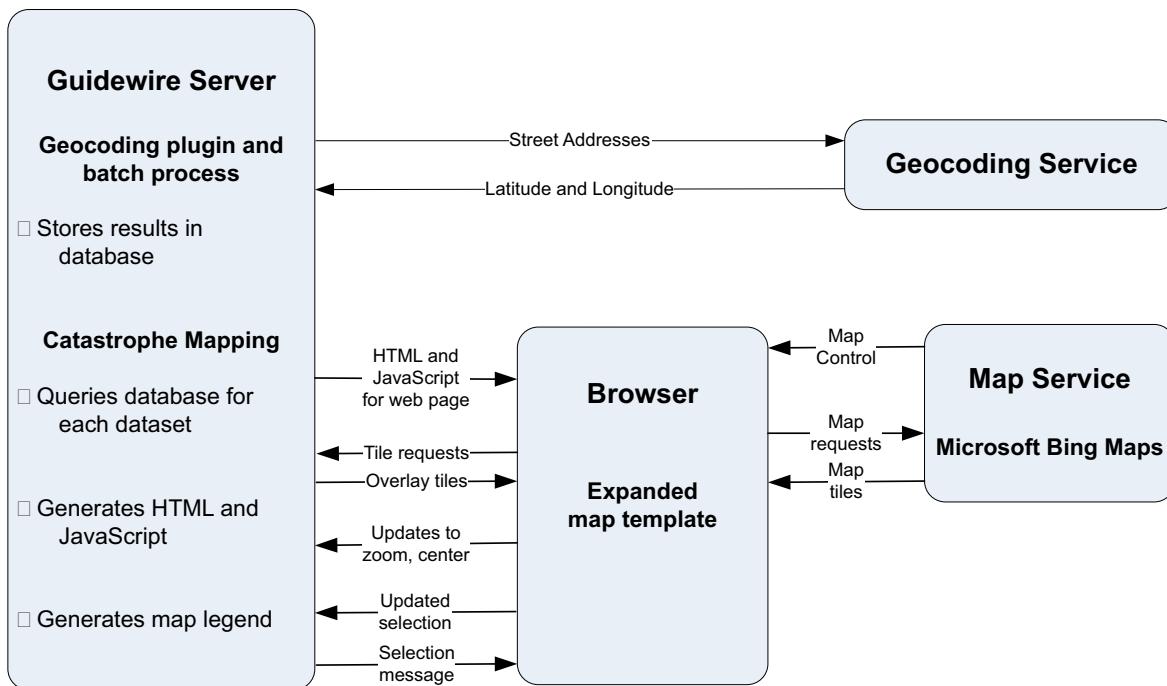
- **Map service** – An external service that provides the map control and map images, such as Bing Maps.
- **Geocoding service** – An external service that provides the latitude and longitude for map points from the street address. Geocoding is done separately through a batch process, which saves latitude and longitude information with each geocoded address.
- **Image generation code** – Code that generates overlay tiles, responds to browser requests, and maintains graphic state information.
- **Data definition code** – Code that defines datasets, queries, and the data associated with latitude and longitude pairs.

For large amounts of data, there is a caching mechanism that reduces the amount of memory used and the load on the database. Caching provides an in-memory buffer for the dataset, one copy per cluster member, which is shared between all users. Without caching, each user has an independent in-memory buffer.

- **HeatMapLegend.gs** – Generates the image for the map legend.
- **Map template** – The map template, such as `BingMap.gs`, used to generate HTML and JavaScript sent to the browser.

Server, Browser, and Service Interactions

The following figure shows the interactions between the Guidewire server, the browser, and the external geocoding and map services. The geocoding operation runs separately from catastrophe mapping.



Locations need to be geocoded before they can be shown on the map. A geocoding service provides the latitude and longitude from the street address, which are saved in the database. In ClaimCenter, a claim preupdate rule, **CPU19000 - Geocode Catastrophe Claims**, schedules geocoding if needed for a claim that is associated with a catastrophe.

When the user requests to view a page containing a heat map, the following steps take place to show the web page:

1. For datasets that use caching, the first reference by any user loads the in-memory cache. Subsequent users reuse the cache entry, one per cluster member. For datasets that do not use caching, the server queries the database and puts the result into a map point buffer specific to that heat map. The buffer can be retained for the life of the session.
2. The server substitutes values into the map template associated with `BingMap.gs`, such as the center point of the map.
3. The server generates map legend images.
4. The server sends the browser the HTML for the screen. The map itself is in an `<i>frame</i>`, which refers to the substituted map template. The map template in turn refers to the map control on the map service website.
5. After the map control is loaded, it requests map and overlay tiles based on the size of the map in the browser, the center point, and the zoom settings. `HeatMapGenerator` creates overlay tiles on demand from the map point data in the buffer.

When the user pans or zooms in the map, the map control requests and displays the appropriate map and overlay tiles. The map template sends the new zoom level and center point to the server immediately, preserving this state information if the user navigates away from the map and later returns.

When the user selects an area on the map by dragging a rectangle, the map template sends the coordinates of the rectangle to the server immediately. `HeatMapGenerator` returns the `SelectionMessage` property, which the map

template displays in the screen in the PCF Input widget with the ID property equal to HeatMapGenerator.SelectionMessageID. If HeatMapGenerator.RefreshUponSelection is true, the page is refreshed to update the PCF file elements.

Principal Heat Map Classes and Files

You can see documentation for these methods and fields in ClaimCenter Studio by opening the class they are in. Alternatively, you can access the Gosu API documentation as described at “Gosu Generated Documentation (‘gosudoc’)” on page 37 in the *Gosu Reference Guide*.

Classes and Template Files Not Specific to the Catastrophe Heat Map

The following classes and template files are available both in ClaimCenter and in PolicyCenter 7.0.3 and later unless designated ClaimCenter only.

- HeatMapGenerator.gs – Base class for defining a heat map. The fully qualified name is gw.api.heatmap.HeatMapGenerator.
- HeatMapDataSet.java – Abstract class defining a dataset, its query, and caching. The fully qualified name is gw.api.heatmap.HeatMapDataSet.
- LatLong.java – Base class that holds the latitude and longitude for a map point. The fully qualified name is gw.api.heatmap.LatLong.
- HeatMapCacheBase.java – Abstract base class to define a cache for a single data set (ClaimCenter only). The fully qualified name is gw.api.heatmap.HeatMapCacheBase.
- HeatMapCacheEntry.java – Interface for a cache entry that has status information and blocking and non-blocking load methods (ClaimCenter only). The fully qualified name is gw.api.heatmap.HeatMapCacheEntry.
- HeatMapCachePlugin.gs – Plugin that initializes caches (ClaimCenter only). The fully qualified name is gw.api.heatmap.impl.HeatMapCachePlugin.
- BingMap.gs – Provides settings to use with Bing Maps. The fully qualified name is gw.api.heatmap.BingMap.
- HeatMapHTML.gst – HTML template for the map control and legend. The fully qualified name is gw.api.heatmap.HeatMapHTML.
- BingMapJavaScript.gst – JavaScript template for passing parameters to BingMap.js. The fully qualified name is gw.api.heatmap.BingMapJavaScript.
- BingMap.js – JavaScript code for the heat map. In ClaimCenter Studio, navigate in the Resources pane to configuration → Web Resources → resources → javascript → heatmap → BingMap.js.
- HeatMapLegend.gs – Generates a map legend. The fully qualified name is gw.api.heatmap.HeatMapLegend.
- HeatMapColorMap.java – Interface or mapping from the count or value for a pixel to a color for a single dataset. Holds the legend labels. The fully qualified name is gw.api.heatmap.HeatMapColorMap.
- RangeColorMap.java – Color map that assigns a range of values to a color. For example 1-1000 is the first color. The fully qualified name is gw.api.heatmap.RangeColorMap.

Classes and Other Files Specific to the Catastrophe Heat Map

The following classes, PCF file, and template file are available only in ClaimCenter. The classes are all in the package gw.api.heatmap. You can navigate to these files in ClaimCenter studio by pressing CTRL+N and entering the file name without the extension. Alternatively, for the class and template files, you can navigate in the Resources pane to configuration → Classes → gw → api → heatmap.

- CatastropheSearchScreen.pcf – The Catastrophe Search screen. To edit this file, in ClaimCenter Studio, navigate in the Resources pane to configuration → Page Configuration (PCF) → search → claims → CatastropheSearch-Screen.

- `CatastropheSearchCriteria.gs` – GUI logic for the Catastrophe Search screen.
- `CatastropheHeatMapView.gs` – Defines map views shown in the Catastrophe Search screen.
- `CatastropheHeatMap.gs` – This subclass of `HeatMapGenerator` is the main class that uses the various classes of the map control. This class renders the map and provides the legend icons and tooltips.
- `CatastropheClaimDataSet.gs` – Subclass of `HeatMapDataSet` that provides claim data for the catastrophe heat map. Uses `CatastropheClaimHeatMapCache` to share the buffer among all users.
- `CatastropheClaimLatLong.gs` – Subclass of `LatLong` representing a single claim on the heat map.
- `CatastropheClaimHeatMapCache.gs` – Subclass of `HeatMapCacheBase` that caches claim information associated with a catastrophe for the catastrophe heat map.
- `CatastrophePolicyLocationDataSet.gs` – Subclass of `HeatMapDataSet` that provides policy location data for the catastrophe heat map. Uses `CatastrophePolicyLocationHeatMapCache` to share the buffer among all users.
- `CatastrophePolicyLocationLatLong.gs` – Subclass of `LatLong` representing a single policy location on the heat map.
- `CatastrophePolicyLocationHeatMapCache.gs` – Subclass of `HeatMapCacheBase` that caches policy location information associated with a catastrophe for the catastrophe heat map.
- `CatastropheHeatMapHTML.gst` – HTML template that replaces `HeatMapHTML.gst` to provide support for two legend icons. Located with the other heat map classes. To see this template, in ClaimCenter studio, navigate in the Resources pane to `configuration` → `Classes` → `gw` → `api` → `heatmap` → `CatastropheHeatMapHTML`.

Datasets and Map Data Points

Heat map data is organized into one or more datasets, which represent points on the map with `LatLong` objects. The base `LatLong` class has only the latitude and longitude. Generally you will want to subclass `LatLong` for each dataset to include additional data for filtering. For example, `CatastropheClaimLatLong.gs` includes the database ID of the claim it represents, total incurred value, and values used for filtering the claims that are shown on the map.

Heat map data points are stored in an in-memory buffer that improves performance for displaying the map. Since the buffer can use a significant amount of memory, the best practice is to keep `LatLong` objects as small as possible. For example, each `CatastropheClaimLatLong` object might be around 100 bytes, so 50,000 of these objects will use 5 megabytes.

Be sure to use database keys such as the claim ID to refer to entities rather than including the entities directly in `LatLong`. Referring to a large object like a claim or a policy directly in a `LatLong` will increase memory usage for the map and make the map point query take longer. Fields that are used only in tooltips or the claim and policy location tables do not need to be part of the `LatLong` object. The heat map code does separate queries using the claim ID or policy location ID to get this data.

To link datasets to your heat map, set `HeatMapGenerator.DataSets` to the list of datasets. Each active dataset uses 256 KB of temporary memory per user when generating overlay tiles.

Datasets and Caching

Buffering for each dataset can be done either with a simple per-user buffer or with a somewhat more complex per-cluster member buffer that runs faster and uses less memory. The catastrophe heat map uses the per-cluster member technique. A non-clustered server will use a single buffer for the dataset.

To use a per-user buffer for a dataset:

1. Put the database query in the `mapPointQuery` method in your subclass of `HeatMapDataSet`.
2. Do not override the `get MapPoints` property.

3. Optionally, set `HeatMapGenerator.MapPointsTimeout`, which has a default of 60 minutes. The buffer is released when the user logs out or when the buffer has not been used for the specified interval. The heat map does a new query the next time the buffer is needed.

To use a per-cluster member buffer for a dataset:

1. Use code similar to `CatastropheHeatClaimDataSet.gs` and `CatastropheClaimHeatMapCache.gs`.
2. In the dataset class, a subclass of `HeatMapDataSet.java`, do the following:
 - Define the query in a static method in the dataset, such as in the method `findClaimsForCatastrophe`.
 - Define `mapPointQuery` to get the value for the cache entry.
 - Define `get MapPoints` to refer to `mapPointQuery`.
 - Do not set `MapPoints` in your code.
3. In the cache class, a subclass of `HeatMapCacheBase.java`, do the following:
 - Set the timeout interval for the cache in the constructor call to `super`. The query is repeated after this interval. If the cache value is not fetched for two intervals, the buffer is released.
 - Define a load method that calls the query method in the dataset class.
 - Define `createAndPreload` to create the cache and optionally preload.
Preloading executes the query and loads the buffer when the server comes up. Otherwise, the first user to reference the cache entry triggers the query, which might cause a long wait. Note that `HeatMapCacheEntry` has status information and both blocking and non-blocking load methods.
4. In `HeatMapCachePlugin.gs`, add a line to the `createCaches` method that calls `createAndPreload` in the cache class.

Datasets and Filtering

The catastrophe heat map provides filters that enable the user to dynamically control the data that is included in the map. The filters include the claim status, reported date, and assigned to group. For quicker updates when the user changes the filter, the catastrophe heat map does the filtering from the map buffer contents rather than making them part of the database query. Using the map buffer contents avoids having to do a new query for each filter change. This approach is essential when using a per-cluster member buffer, since all the users on the cluster member share the same buffer but can have different filter settings.

To use or add filters in a dataset:

1. Add the values needed for filtering to your `LatLong` subclass. To keep memory use down and make the filter condition evaluate quickly, you can precompute complex filter conditions and reduce them to simple variables saved in the `LatLong` subclass.
2. Add the necessary data to the query for your dataset. If you are using an Oracle materialized view for the query, you must update the materialized view definition.
3. Define a filter method in the dataset class. Because the filter is evaluated very frequently, make the function as simple and quick as possible. Avoid lengthy loops and database queries.

DBA Scripts to Improve Oracle and SQL Server Performance

The scripts described in this section give better performance for the query that loads claims into the per-cluster member cache. Ask your DBA to apply them.

Oracle Script

You can find a copy of the following sample Oracle script at the following location:

`modules/p1/deploy/oracle/catmapmv.sql`

The script creates a materialized view for Oracle. You must set some parameters on the claims query to take advantage of the materialized view. Setting these parameters is similar to setting up the query parameter of the `enableMaterializedView` method in `CatastropheClaimDataSet.findClaimsForCatastrophe`.

```

create materialized view log on cc_claim initrans 8 tablespace CC_OP with PRIMARY KEY (
    CatastropheID,retired, state, AssignedGroupID, reportedDate),
    rowid including new values;
create materialized view log on cc_address initrans 8 tablespace CC_OP with PRIMARY KEY (
    retired, latitude, longitude), rowid including new values;
create materialized view log on cc_claimrpt initrans 8 tablespace CC_OP with PRIMARY KEY (
    ClaimID, retired, OpenReservesRprtng, TotalPaymentsRprtng, TotalRecoveriesRprtng),
    rowid including new values;

alter session set "_optimizer_compute_index_stats"=false;

create index i_mlog_claim on MLOG$_CC_CLAIM(XID$$, M_ROW$$)
    tablespace cc_index initrans 8;
create index i_mlog_address on MLOG$_CC_ADDRESS(XID$$, M_ROW$$)
    tablespace cc_index initrans 8;
create index i_mlog_claimrpt on MLOG$_CC_CLAIMRPT(XID$$, M_ROW$$)
    tablespace cc_index initrans 8;

exec dbms_stats.UNLOCK_TABLE_STATS(user, 'MLOG$_CC_CLAIM');
exec dbms_stats.UNLOCK_TABLE_STATS(user, 'MLOG$_CC_ADDRESS');
exec dbms_stats.UNLOCK_TABLE_STATS(user, 'MLOG$_CC_CLAIMRPT');
exec dbms_stats.gather_table_stats(user, 'MLOG$_CC_CLAIM', cascade=>true);
exec dbms_stats.gather_table_stats(user, 'MLOG$_CC_ADDRESS', cascade=>true);
exec dbms_stats.gather_table_stats(user, 'MLOG$_CC_CLAIMRPT', cascade=>true);
exec dbms_stats.lock_table_stats(user, 'MLOG$_CC_CLAIM');
exec dbms_stats.lock_table_stats(user, 'MLOG$_CC_ADDRESS');
exec dbms_stats.lock_table_stats(user, 'MLOG$_CC_CLAIMRPT');

CREATE materialized view catmapmv
tablespace cc_op initrans 4
USING INDEX tablespace cc_index initrans 4
refresh fast on commit
-- start with sysdate next sysdate + 1/48
WITH Primary Key
enable query rewrite
AS
SELECT
    c.ID ClaimID, c.State ClaimState, c.AssignedGroupID ClaimAssignedGroupID,
    c.ReportedDate ClaimReportedDate, c.CatastropheID CatastropheID, c.rowid ClaimRowID,
    c.Retired ClaimRetired, a.Retired AddressRetired, a.ID AddressID,
    a.rowid AddressRowID, a.Latitude AddressLatitude, a.Longitude AddressLongitude,
    r.rowid ClaimRptRowID, r.ID ClaimRptID, r.Retired ClaimRptRetired,
    r.ClaimID ClaimRptClaimID, r.OpenReservesRprtng OpenReserves,
    r.TotalPaymentsRprtng TotalPayments, r.TotalRecoveriesRprtng TotalRecoveries
FROM
    cc_claim c, cc_address a, cc_claimrpt r
WHERE
    a.ID = c.LossLocationID AND c.ID = r.ClaimID;

create index i_catmapmv on catmapmv(CATASTROPHEID, CLAIMRETIRED, ADDRESSRETIRED,
    CLAIMRPTRETIRED, ADDRESSLATITUDE, ADDRESSLONGITUDE, CLAIMID, CLAIMSTATE,
    CLAIMASSIGNEDGROUPID, CLAIMREPORTEDDATE, OPENRESERVES, TOTALPAYMENTS,
    TOTALRECOVERIES) tablespace cc_index initrans 3;

create unique index rowidc on catmapmv(ClaimRowID) tablespace cc_index initrans 2;
create unique index rowida on catmapmv(AddressRowID) tablespace cc_index initrans 2;
create unique index idc on catmapmv(ClaimID) tablespace cc_index initrans 2;
create unique index ida on catmapmv(AddressID) tablespace cc_index initrans 2;
create unique index rowidr on catmapmv(ClaimRptRowID) tablespace cc_index initrans 2;

exec dbms_stats.gather_table_stats(user, 'catmapmv', cascade => true,
    estimate_percent => dbms_stats.auto_sample_size);

```

SQL Server Script

You can find a copy of the following sample SQL Server script at the following location:

`modules/pl/deploy/sqlserver/catmapindexview.sql`

The script creates an indexed view for SQL Server:

```

IF EXISTS (SELECT *
    FROM  dbo.sysobjects
    WHERE id = Object_id(N'[dbo].[catmapiv]')
        AND Objectproperty(id, N'IsView') = 1)

```

```
DROP VIEW [dbo].[catmapiv]
GO

--Index view for catastrophe heat map queries
CREATE VIEW [dbo].catmapiv
WITH SCHEMABINDING
AS
    SELECT c.id          claimid,
           c.state       claimstate,
           c.assignedgroupid   claimassignedgroupid,
           c.reporteddate   claimreporteddate,
           c.catastropheid  catastropheid,
           c.retired       claimretired,
           a.retired       addressretired,
           a.id           addressid,
           a.latitude     addresslatitude,
           a.longitude    addresslongitude,
           r.id           claimrptid,
           r.retired      claimrptretired,
           r.claimid      claimrptclaimid,
           r.openreservesrprtn openreserves,
           r.totalpaymentsrprtn totalpayments,
           r.totalrecoveriesrprtn totalrecoveries
    FROM [dbo].cc_claim c,
         [dbo].cc_address a,
         [dbo].cc_claimrpt r
   WHERE a.id = c.losslocationid
     AND c.id = r.claimid;
GO

CREATE UNIQUE CLUSTERED INDEX i_catmapmv
    ON catmapiv(catastropheid, claimretired, addressretired, claimrptretired,
                  addresslatitude, addresslongitude, claimid, claimstate,
                  claimassignedgroupid, claimreporteddate,
                  openreserves, totalpayments, totalrecoveries);
GO

CREATE UNIQUE INDEX idc
    ON catmapiv(claimid);
GO

CREATE UNIQUE INDEX ida
    ON catmapiv(addressid);
GO
```

Configuring the Heat Map

This topic presents some of the configuration points you can use to configure the heat map.

This topic includes:

- “Common Configuration Use Cases” on page 545
- “Advanced Configuration” on page 545
- “Bing Maps Limitations” on page 546
- “Working with a Different Map Service” on page 546
- “Ajax-style Extensions to the Heat Map” on page 546

Common Configuration Use Cases

The following table lists some typical configurations and describes the configuration points for each one.

Configuration	How to do it
Add a map view.	In <code>CatastropheHeatMapViewViews.gs</code> : 1. Define a new instance of <code>CatastropheHeatMapView</code> similar to <code>ClaimCountsView</code> . 2. Add the view name to <code>AvailableViews</code> . If the view shows claims or policies or both, add the view name to <code>ClaimViews</code> or <code>PolicyLocationViews</code> or both.
Show additional data in the policy location table.	Modify the query in <code>CatastropheSearchCriteria.performHeatMapPolicyLocationSearch</code> and <code>PolicyLocationSearchResultsLV.pcf</code> .
Modify tooltips.	Modify the code in <code>getToolTip</code> in <code>CatastropheHeatMap.gs</code> . The tooltip is represented as an HTML fragment.
Modify the selection message.	The heat map shows the selection message when the user set the selection rectangle on the map, such as, “56 claims and 112 policy locations selected”. The selection message varies depending on the map view. See the code for each map view in <code>SelectionMessage</code> in <code>CatastropheHeatMapViewViews.gs</code> .
Change heat map colors and legend labels.	In <code>CatastropheHeatMapViewViews.gs</code> , pass the heat map colors to the constructor for <code>RangeColorMap</code> , which initializes the labels appropriately. You can change the labels after calling the constructor, as is done for <code>policyColorMap</code> . Each dataset is associated with one color map.
Show amounts on the map rather than counts.	In the dataset class, define a <code>getWeight</code> method that returns the amount. The amount must be an <code>int</code> and greater than or equal to zero. To show a marker on the map for a zero value, subclass <code>RangeColorMap</code> and override <code>getColorForValue</code> as is done in <code>AmountColorMap</code> in <code>CatastropheHeatMapViewViews.gs</code> .
Set the map size.	You can change the height by changing the <code>setSize</code> call in the constructor for <code>CatastropheHeatMap.gs</code> . The HTML code specifies using all the available width. To use a specified width instead, change <code>CatastropheHeatMapHTML.gst</code> , replacing the string “width:100%” in two places with “width:\${ requestedWidth }px”.
Other configurations.	Refer to the Gosu API reference for the relevant classes, as described at “Gosu Generated Documentation (‘gosudoc’)” on page 37 in the <i>Gosu Reference Guide</i> .

Advanced Configuration

The configuration techniques described in this topic are advanced and are not a typical part of configuration. They are included in case you need to perform these particular configurations.

This topic includes:

- “Extending the JavaScript” on page 545
- “Bing Maps Limitations” on page 546
- “Working with a Different Map Service” on page 546
- “Ajax-style Extensions to the Heat Map” on page 546

Extending the JavaScript

`BingMap.js` is organized to enable you to extend it without modifying `BingMap.js` itself. For example, you can use additional Bing capabilities such as showing push-pin markers or displaying a mini-map within the main map.

To extend `BingMap.js`:

1. Create a new JavaScript file in the same directory as `BingMap.js`. You can subclass `HeatMap` and override methods by following JavaScript conventions. Structure your file like this:

```
function MyExtension() { }
MyExtension.prototype = new HeatMap();
MyExtension.prototype.superclass = HeatMap;
MyExtension.prototype.constructor = MyExtension;
```

```
// put your code here, such as:  
MyExtension.prototype.MyFunction() {  
    // your code here  
};  
  
heatMap = new MyExtension();
```

- 2.** Subclass `BingMaps.gs` and override `javaScriptFileNames` to include your JavaScript file after the `BingMaps.js` file. Change the `HeatMapServiceTemplate` parameter in `config.xml` to use the fully-qualified subclass name.

Bing Maps Limitations

The Bing Maps control puts the International Date Line only at the left or right edge of the map. The date line cannot be in the middle. This requirement has two effects:

- If the set of map points spans the date line, the points will be split up in an inconvenient or even misleading way.
- On large scale maps, the requested center point, such as from computing a bounding box for the map points, might not appear in the center of the map.

Like most map services, Bing Maps is limited to showing points with latitudes between roughly -86 to +86 degrees, which provides a square map of the world. The map projection from the sphere of the earth onto a plane puts the north and south poles at infinity, which is impractical to display.

Working with a Different Map Service

You might be able to interface with other mapping services by replacing `BingMap.gs`, `BingMap.js`, and `BingMapJavaScript.gst`.

`HeatMapGenerator` assumes that map tiles are 256 x 256 pixels in size. It accepts overlay tile requests that use Bing- or Google-style tile identifiers. Your code replacing `BingMap.gst` would need to return one of these types of tile identifiers. The `tile` parameter in the HTTP request identifies which tile to return, as shown in the following table:

Map Service	Example	General Form
Bing	0321.png	[0-3]+\png
Google	tile_9_21_35.png	tile_(0-9+)_([0-9]+_)_(0-9+)\.png

Bing uses quad tree-based keys. At the lowest zoom level, the entire world, the map is divided into four tiles, each identified by a single-digit number, 0..3. At the next zoom level, each tile is subdivided into four subtiles, identified as 00, 01, 02, 03, 10, 11, 12, 13, and so on. See <http://msdn.microsoft.com/en-us/library/bb259689.aspx> for a full description.

The numbers in the Google identifier are the zoom level, the x coordinate, and the y coordinate. See the following web pages for a full description:

- http://code.google.com/apis/maps/documentation/javascript/v2/overlays.html#Tile_Overlays
- http://code.google.com/apis/maps/documentation/javascript/v2/overlays.html#Custom_Map_Types

You might find it helpful to use Firefox and the Firebug debugger to develop the interface to a new map service. Firebug can display all messages sent to and from the browser.

Ajax-style Extensions to the Heat Map

By overriding `HeatMapGenerator.handleRequest` and modifying or extending `BingMap.js`, you can add additional Ajax-style interactions between the browser and the server to the heat map. See the `handleRequest` method in `CatastropheHeatMap.gs` for an example. Be sure to include a call to `super.handleRequest`. See the

Gosu API reference for information on `HeatMapGeneratorBase.handleRequest`. To use this reference, see “Gosu Generated Documentation (‘gosudoc’)” on page 37 in the *Gosu Reference Guide*.

To send UTF-8 encoded text, use code like this:

```
 servletResponse.setHeader("Content-Type", "text/plain")
 // use StreamUtil.toBytes() to handle UTF-8 characters correctly
 servletResponse.getOutputStream().write(StreamUtil.toBytes(TextMessage))
```


Configuring Duplicate Claim and Check Searches

This topic explains how to configure the Gosu templates so that you can modify the search criteria for duplicate claims and checks. ClaimCenter checks if there are any matching claims or checks to avoid duplication.

This topic includes:

- “Understanding the Gosu Templates” on page 549
- “Duplicate Claim Search” on page 550
- “Duplicate Check Search” on page 551

Understanding the Gosu Templates

You can modify the search criteria for duplicate claims and duplicate checks in the Gosu templates in Studio. Navigate to **configuration → gsrc → gw → duplicatesearch**. The folder contains the following templates:

Gosu template	Description
gw.duplicatesearch.DuplicateCheckSearchTemplate	Duplicate Check search
gw.duplicatesearch.DuplicateClaimSearchTemplate	Duplicate Claim search

Parameters

The `DuplicateCheckSearchTemplate` takes three parameters:

- A `DuplicateSearchHelper`, which provides utility methods for SQL construction.
- The check to search for duplicates.
- A `checkBeingCloned` parameter. If the Check is a clone of an existing check, this parameter contains the existing Check. The search avoids returning the existing Check or any of its recurrences as a duplicate. Otherwise, `checkBeingCloned` is `null`.

The `DuplicateClaimSearchTemplate` takes just two parameters:

- A `DuplicateSearchHelper`, which provides utility methods for SQL construction.
- The `Claim` to search for duplicates.

Gosu Language

The following table displays how the template uses the Gosu language.

Area	Gosu
Comments	<pre>/* This is a comment that spans multiple lines. */ and // This is a single-line comment.</pre>
Initializing a variable	<code>var myVar = 123</code>
Modifying a variable	<code>myVar = "new Value"</code>
Conditional expressions	The condition has to be a Boolean or the template does not compile. It is possible for a Boolean to be <code>null</code> , in which case it is treated as <code>false</code> .
If conditional block	<code>if (myCondition) { ... }</code>
If-else conditional block	<code>if (myCondition) { ... } else { ... }</code>

Duplicate Claim Search

In the base configuration, the Gosu template creates the SQL query for finding duplicate claims. If it finds any that match the query, the user interface displays a list of claim IDs that match the claim that is being created. This allows the user to cancel the claim, if needed. If there are no matches, the user does not see any messages.

The query considers the claim to be a duplicate if one of the following conditions is satisfied.

- The claim has the same policy and has a loss date that is within +/- 3 days of the claim's loss date, or
- The claim's insured has the same name and the loss date is within +/- 3 days of the claim's loss date. If there is no name, then the query searches for the company name.

To change the search criteria

You can modify `DuplicateClaimSearchTemplate` or add any Gosu code to extend or modify the search criteria. For example, you can change the length of time that the system checks for a duplicate claim by doing the following:

1. In `DuplicateClaimSearchTemplate`, find the following functions:

```
function genClaimLossDateThreeDaysPriorParameter() : String {
    return helper.makeParam("Claim.LossDate", claim.LossDate.addDays(-3))
}
and
function genClaimLossDateThreeDaysAfterParameter() : String {
    return helper.makeParam("Claim.LossDate", claim.LossDate.addDays(3))
```

2. Change the numeric value in bold, save your work, and restart the application server.

Duplicate Check Search

The Gosu template `DuplicateCheckSearchTemplate`, which is used to create the SQL query for finding duplicate checks, verifies that the check has not already been created. If it finds any duplication, the user interface displays a list of check IDs that match the check. Otherwise, the user does not see a message.

In the base configuration, the SQL query looks for checks written to the same person for the same amount on the claim.

Check B is considered to be a duplicate of check A if:

- Check A and check B have the same `PayTo` field *or* check B has a payee with the same `TaxID` as one of the payees on check A.
- Check A and check B are on the same claim.
- Check A and check B have the same gross amount.
- Check A and check B have the same currency.
- The service periods (`ServicePdStart` to `ServicePdEnd`) for check A and check B overlap *or* both check A and check B have incomplete service periods (one or both fields are `null`)
- If Check A is created as a clone of another check, Check B must not be the check from which A is being cloned (or of any of its recurrences). This is how the `checkBeingCloned` argument is used.

If there are multiple payees, then the query looks at all contacts (through claim contact) that have the `checkpayee` role (`id=10011`). It next searches on the contact tax ID list passed in to the template.

To change the search criteria

Similar to `DuplicateClaimSearchTemplate`, you can customize `DuplicateCheckSearchTemplate` template to extend or modify the search criteria.

Configuring Claim Health Metrics

This topic describes how to configure Claim Health Metrics. You can add a new tier, a high-risk indicator, or a new claim metric. You can also add a new tier for additional granularity, like exposure tiers, which are more granular than claim tiers. You can create a high risk indicator for anything that is important to your business. For example, you might add a high-risk indicator for property damage over a certain amount or perhaps a missed doctor's appointment for the workers' compensation policy type. You might also create a new metric to measure time to get an estimate complete for the personal auto policy type.

This topic includes:

- “Adding a New Tier” on page 553
- “Adding a High-Risk Indicator” on page 555
- “Adding a New Claim Metric” on page 558

See Also

- “Claim Performance Monitoring” on page 391 in the *Application Guide* to learn about this feature.
- “Managing Metrics and Thresholds” on page 500 in the *Application Guide* to learn how to administer claim health metrics.

Adding a New Tier

To add a new tier, create it first in Studio, then define its associated limit values in the ClaimCenter Administration tab user interface, and finally implement it with Gosu logic.

Perform the following steps to add a new tier.

To associate a new tier value to a typelist in Studio

1. You must first add a new tier value to the typelist. Choose either the *ClaimTier* or *ExposureTier* typelist. Navigate in Studio to configuration → Typelists.

The ClaimTier and ExposureTier typelists define the tiers, as shown in the following example:

Code	Name	Description	Priority	Retired
indemnity	Indemnity	A claim with an indemnity i	300	false
low	Low Severity	Low Severity	100	false
medicalonly	Medical Only	A claim with a medical co	200	false

TypeList	Code
PolicyType	InlandMarine
PolicyType	PersonalAuto
PolicyType	PersonalUmbrella

Each tier is associated with **PolicyTypes** through **Categories**. In this example, notice that the claim tier value of *Low Severity* is associated with the personal auto policy type (as well as with others.)

2. Click **Add** in the **Code** tab and enter the new tier value.
3. Associate the new tier value with the selected **PolicyTypes** using the **Categories** tab. Do this by clicking **Add** in the **Categories** tab and adding the policy type.
4. Save your work and exit Studio.

To define target values for the new tier in ClaimCenter

Repeat this process for each metric and each policy type for which you want to set metric limits.

1. Navigate to the **Administration** tab → **Metrics and Thresholds**.
2. Choose the policy type for the metric limit you want to set.
3. Click **Edit** and select the new tier from drop-down list for the metric whose limit you are setting.
4. Enter the target values for the new tier on that metric and policy type.
5. Click **Update**.

To edit the tier enhancement Gosu code in Studio

Now that you have created a tier, you must add Gosu logic to set your new tier on claims and exposures. Tiers are calculated as part of the claim health update process, which updates high risk indicators, sets tiers, and updates metrics. This process happens after pre-update rules are executed. The enhancement methods `setClaimTier` and `setExposureTier` are called to update the claim and exposure tiers. You must alter these enhancement methods to set your new tier if the conditions are right. Without editing `GWClaimTierEnhancement` or `GWExposureTierEnhancement`, new tiers cannot be assigned.

In Studio, edit the Gosu code to add logic to assign new tier values to claims or exposures.

1. Navigate in Studio to **configuration** → **Classes** → **gw** → **entity**.
2. Select either the `GWClaimTierEnhancement` or `GWExposureTierEnhancement` enhancement file.

3. Edit the enhancement file to add logic for assigning new tier values. Current logic assigns tier values based on various factors:
 - Where in line of business hierarchy the claim or exposure falls
 - Incident subtype
 - Complexity of entity
 - Severity of incident
 - Is the claim in litigation
 - Is a vehicle a total loss
 - Was there a fatality

The logic might make decisions based on `PolicyType`, `CoverageType`, `CoverageSubType`, or `ExposureType`.

Adding a High-Risk Indicator

You can create a new high-risk indicator. The general steps are that you must create the icon, create a subtype of `ClaimIndicator`, and add the implementation code.

WARNING Guidewire strongly recommends limiting the number of indicators used for any one line of business. Overuse of indicators lessens the overall impact to end users. Additionally, Guidewire designed the **Claim Summary** screen with the expectation that few, if any, claims will have more than four or five indicators. If the number of indicators per claim exceeds this expectation, Guidewire recommends that you revisit the **Claim Summary** screen design and determine if it needs to be modified. Otherwise, important claim information can appear farther down the screen, necessitating additional scrolling.

Note: Each time you create a new subtype, you must modify the PCF files to show the new indicator in the info bar and on the **Claim Status** screen. If the indicator implements the standard `On` property and has an icon, it can also appear on the **Claim Summary** screen. That screen relies on the generic indicator interface and can show the indicator automatically, whenever it is on.

To create the high-risk icon

1. Create the image using a third-party graphics program.
2. In Guidewire Studio, navigate to **configuration** → **Web Resources** → **resources** → **Ocean** → **images**. Right-click and select **New** → **Other file**. Enter the same file name as the name of the file created in the third-party graphics program.
3. Copy the third-party graphics image file into the install directory and replace the file created by Studio. The path is `ClaimCenter\modules\configuration\webresources\Ocean\images`.
4. If the server is running, stop it.
5. Execute `gwcc dev-deploy` in the Command window.
6. Restart the server.

To add a new subtype in Guidewire Studio

1. You extend the ClaimCenter data model by adding a new subtype of the entity `ClaimIndicator`. In Studio, navigate to **configuration** → **TypeLists** → **ClaimIndicator** to see the following subclasses listed under the **Code** column:
 - `ClaimIndicator`
 - `CoverageInQuestionClaimIndicator`
 - `FatalityClaimIndicator`
 - `LargeLossClaimIndicator`

- LitigationClaimIndicator
 - FlagClaimIndicator
 - SIUClaimIndicator
2. Define a new entity subtype. The **supertype** you choose depends on the type of indicator that you want. For example:
- a. Navigate to **configuration** → **Data Model Extensions** → **extensions** and right-click **extensions**.
 - b. Click **New** → **Other** file.
 - c. Enter the file name **ExampleClaimIndicator.eti** and click **OK**.
 - d. Enter the following entity definition:
- ```
<?xml version="1.0"?>
<subtype xmlns="http://guidewire.com/datamodel" desc="Litigation"
 entity="ExampleClaimIndicator" final="false" priority="1"
 supertype="ClaimIndicator">
<implementsInterface
 iface="gw.api.claim.indicator.ClaimIndicatorMethods"
 impl="gw.claim.indicator.ExampleClaimIndicatorMethodsImpl"/>
</subtype>
```
- e. Save the file.
- The **iface** attribute defines the interface that the subtype implements. The **implementsInterface** element must be present, and the **impl** class must be your own implementation class. Use the package **gw.claim.indicator** and your new **SubtypeNameMethodsImpl** class, which you define later.
3. Stop and restart Studio to automatically create a typecode for the new subtype to the **ClaimIndicator** type-list. You configure the secondary attributes of the typecode—**Name**, **Description**, and **Priority**. For example, navigate to **configuration** → **Typelists** → **ClaimIndicator** and click the new typecode to edit it. If you see a message asking if you want to edit the typelist, click **Yes**.

| Code                  | Name    | Description             | Priority | Retired |
|-----------------------|---------|-------------------------|----------|---------|
| ExampleClaimIndicator | Example | Example Claim Indicator | 4        | false   |

This step enables you to set up a possibly localized name and priority for the indicator. The **Name** attribute is the name you want to show in the user interface for this type of indicator. The **Priority** determines the order in which the indicators appear on the screen.

4. Implement your new **SubtypeNameMethodsImpl** class. Your class can implement the **ClaimIndicatorMethods** interface. If, instead, you extend the **ClaimIndicatorMethodsImpl** class, you get the following conveniences:
- Automatic handling of the icon. You need to specify only the string name of your indicator icon when you call the constructor of **ClaimIndicatorMethodsImpl**.
  - The method **setOn(newValue: boolean)**, which you can use to set the **IsOn** flag and the **WhenOn** field of the indicator.

Following is an example of an implementation:

```
package gw.claim.indicator
uses gw.api.claim.indicator.ClaimIndicatorMethodsImpl
class ExampleClaimIndicatorMethodsImpl extends ClaimIndicatorMethodsImpl {
 /**
 * Constructor, called when an indicator is created or read from the database
 */
 construct(inIndicator : ExampleClaimIndicator) {
 super(inIndicator, "indicator_icon_litigation.gif") // Passes in name of indicator icon
 }
 /**
 * Update, sets the indicator on if the the claim litigation status is "litigated"
 * or "complete"
 */
 override function update() {
 var status = Indicator.Claim.LitigationStatus
 setOn(status == "litigated" or status == "complete")
 // Calls setOn to set both IsOn and WhenOn fields
 }
}
```

```

 }
 /**
 * Text label, returns the description of the current claim litigation status
 */
 override property get Text() : String {
 return Indicator.Claim.LitigationStatus.Description
 }
 /**
 * Hover text returns the names of any open matters,
 * or a special label if there are none.
 */
 override property get HoverText() : String {
 var openMatters = Indicator.Claim.Matters.where(
 \ m -> not m.Closed).orderBy(\ m -> m.CreateTime)
 return openMatters.Count > 0
 ? openMatters.map(\ m ->
 m.Name).join(displaykey.Web.LitigationClaimIndicator.MatterNameSeparator)
 : Indicator.Claim.LitigationStatus.DisplayName
 }
}

```

5. After implementing the class required by your claim indicator entity, you can regenerate the data dictionary to ensure that the new entity has the correct definition.

At a command prompt, navigate to `ClaimCenter\bin` and enter:

```
gwcc regen-dictionary
```

6. Add a new info bar element to the `ClaimInfoBar` PCF file. You must add the element *explicitly*, even though the required element is standard. For example, for a new indicator type called `ExampleClaimIndicator`:
- Navigate to **configuration** → **Page Configuration (PCF)** → **claim** → **ClaimInfoBar**.
  - Drag an **InfoBarElement** from the **Toolbar** on the right and drop it on the **InfoBar**. If you see a message asking if you want to edit the file, click **Yes**.
  - Click the new **InfoBarElement** and set the following properties:

| Property             | Value                                              |
|----------------------|----------------------------------------------------|
| <code>id</code>      | <code>ExampleClaimIndicator</code>                 |
| <code>icon</code>    | <code>Claim.ExampleClaimIndicator.Icon</code>      |
| <code>tooltip</code> | <code>Claim.ExampleClaimIndicator.HoverText</code> |
| <code>visible</code> | <code>Claim.ExampleClaimIndicator.IsOn</code>      |

7. Add a new modal input set called `ClaimIndicatorInputSet.ExampleClaimIndicator.pcf`.

The **Claim Status** screen uses this input set to display the details of your indicator. This screen iterates through all indicators on the claim and displays an input set for each indicator. It handles the indicator as a *require* argument to the input set. You can put whatever you want in the input set. It is helpful to follow a style similar to that of the input sets for the existing indicators, which tend to be fairly small. If you make your input set too large, it can drastically change the layout of the **Claim Status** screen.

ClaimCenter displays the indicator input sets by subtype order. An indicator with subtype set to priority 1 appears before an indicator with subtype set to priority 3.

For example:

- Navigate to **configuration** → **Page Configuration (PCF)** → **claim** → **summary** → **indicator** → **ClaimIndicatorInputSet.LitigationClaimIndicator**.
- Right-click `ClaimIndicatorInputSet.LitigationClaimIndicator` and click **Duplicate**.
- Name the new PCF file `ClaimIndicatorInputSet.ExampleClaimIndicator.pcf`. If you see a message asking if you want to create a copy of the folder, click **Yes**.
- Click `ClaimIndicatorInputSet.ExampleClaimIndicator` to open it in the editor.

You now have a new input set with a set of widgets. You can modify this input set to meet your needs.

## Adding a New Claim Metric

You can create a new claim metric. The general steps are to create a subtype of an existing metric and then add the implementation code.

You do not need to add the metric to the typelist. This addition occurs automatically. If you add a new subtype, the platform layer automatically adds a new member to the associated typelist. For example, if you add a new subtype of `ClaimMetric`, the system adds a new member to the `ClaimMetric` typelist. Adding a new subtype is the *only* way of adding new members to this typelist.

### To add a new claim metric in Guidewire Studio

1. Extend the ClaimCenter data model by adding a new subtype of the entity `ClaimMetric`. One way to do this is to extend one of the pre-supplied claim metric classes. In Studio, navigate to **configuration** → **Data Model Extensions** → **metadata** → `cc` to see the following claim metric classes:

- `DecimalClaimMetric.eti`
- `IntegerClaimMetric.eti`
- `MoneyClaimMetric.eti`
- `PercentClaimMetric.eti`
- `TimeBasedClaimMetric.eti`

Your selection depends on the type of quantity the metric is tracking.

**Note:** You can add a direct implementation of `ClaimMetric` rather than subtyping one of the pre-supplied claim metric classes. Doing so is appropriate if the value of the metric does not fall into any of the pre-supplied types—integer, decimal, percent, money or time based. It is also possible to add arbitrary new data fields to your subtype if your metric needs them.

2. You can create a new metric based on `TimeBasedMetric` as follows:

- a. Navigate to **configuration** → **Data Model Extensions** → **extensions** and right-click **extensions**.
- b. Enter the file name `ExampleClaimMetric.eti` and click **OK**.
- c. Enter the following entity definition. If you copy and paste the following code, delete any leading spaces before the first line of code.

```
<?xml version="1.0"?>
<subtype desc="Example Claim Metric" entity="ExampleClaimMetric"
 final="false" priority="1" supertype="TimeBasedClaimMetric">
 <implementsInterface
 iface="gw.api.metric.MetricMethods"
 impl="gw.claim.metric.general.ExampleClaimMetricMethodsImpl"/>
 <implementsInterface
 iface="gw.api.claim.metric.RecalculateMetrics"
 impl="gw.claim.metric.general.ExampleClaimMetricMethodsImpl"/>
</subtype>
```

- You must use the first `implementsInterface` element and specify your own implementation class for the `impl` attribute. The implementation class name is, by convention, `SubtypeNameMethodsImpl`.
- The second `implementsInterface` element makes it possible for the Recalculate Claim Metrics batch job to recalculate this metric. You rarely need to implement `ReCalculateMetrics`. If you do so, you must also implement this interface in the class you specify in the `impl` attribute, which is the same class in the previous `implementsInterface` element. For more information, see step 6.

- d. Save the file.

3. Close Studio, and then restart it to automatically add a typekey for your new subtype to the `ClaimMetric` typelist.

4. You can configure the new typecode by navigating to **configuration** → **Typelists** → `ClaimMetric`.

**Note:** If you do not see the new typecode in the `ClaimMetric` typelist, there is probably an error in your entity definition. Check your definition and make sure that it is correct.

5. Click the new typecode to edit it. If you see a message asking if you want to edit the file, click Yes. Set the following values:

| Code               | Name    | Description          | Priority | Retired |
|--------------------|---------|----------------------|----------|---------|
| ExampleClaimMetric | Example | Example Claim Metric | 4        | false   |

The **Name** of the typecode element is the name that appears in the user interface for this type of metric. You can localize this name. The **Priority** determines the ordering of the metric in the Claim Metrics user interface. It appears after any metrics with priority less than 4 and before any with priority more than 4.

6. Extend one of the following classes:

- gw.api.claim.metric.DecimalClaimMetricMethodsImpl
- gw.api.claim.metric.IntegerClaimMetricMethodsImpl
- gw.api.claim.metric.MoneyClaimMetricMethodsImpl
- gw.api.claim.metric.PercentClaimMetricMethodsImpl
- gw.api.claim.metric.TimeBasedClaimMetricMethodsImpl

Since you are subtyping from one of the entities listed in step 1, you can extend a matching Gosu class, which does a lot of the work for you. For the example in step 2, you need to implement TimeBasedClaimMetricMethodsImpl.

If your metric does not match any of these classes, you can extend the MetricMethodsImpl class. To see this class, navigate to configuration → Classes → gw → api → metric → MetricMethodsImpl.

**Note:** If your metric requires periodic recalculating, you must implement gw.api.claim.metric.RecalculateMetrics. Implementing this interface makes it possible for the Recalculate Claim Metrics batch job to run on this metric object. For example, a metric that provides a count of overdue activities could require recalculating if existing activities become overdue. In this case, there is no database update to trigger the update method, so recalculation happens only if the batch job runs.

7. When you extend one of the claim metric classes, you must create a class with a constructor and an override of the updateMetricValue method. The following class extends the TimeBasedClaimMetricMethodsImpl class and, additionally, implements the RecalculateMetrics interface. When you implement this interface, you must also override the recalculate method:

```
package gw.claim.metric

uses gw.api.claim.metric.TimeBasedClaimMetricMethodsImpl
uses gw.api.metric.MetricUpdateHelper
uses gw.api.claim.metric.RecalculateMetrics
uses java.util.Date
@Export
class ExampleClaimMetricMethodsImpl extends TimeBasedClaimMetricMethodsImpl
 implements RecalculateMetrics {
 construct(exampleClaimMetric : ExampleClaimMetric) {
 super(exampleClaimMetric, ClaimMetricCategory.TC_OVERALLCLAIMMETRICS)
 }
 override function updateMetricValue(helper : MetricUpdateHelper) : Date {
 Metric.StartTime = Metric.Claim.ReportedDate
 handleClaimStateChange()
 return null
 }
 override function recalculate() : Date {
 var result = recalculateValue()
 updateMetricLimitReachedTimes()
 return result
 }
}
```

This example is time-based. It extends the provided TimeBasedClaimMetricMethodsImpl class, which gives access to time specific fields like Metric.StartTime. The class also provides the handleClaimStateChange method for updating the metric state if the claim opens or closes. Additionally, the recalculate method

updates the limit reached times if the new metric value has caused it to pass a limit. The method returns the time when you want to metric to be recalculated.

**Note:** This example is somewhat artificial, because as long as the metric is open, its value changes with time. You typically use RecalculateMetrics with a metric that is not time-based and whose value might change due to some known future event that does not affect the database. For example, a metric that provides a count of overdue activities must be recalculated by the batch job when an existing activity becomes overdue.

---

**IMPORTANT** All implementation classes must have a constructor that takes one parameter of the actual metric type. The `implementsInterface` mechanism requires this constructor because the object is created whenever the metric is read from the database.

---

8. If you need to do something only when the metric is first created, you can add a constructor like the following one:

```
construct(exampleClaimMetric : ExampleClaimMetric) {
 super(exampleClaimMetric, ClaimMetricCategory.TC_OVERALLCLAIMMETRICS)
 if (exampleMetric.New) {
 // Do your initialization here
 }
}
```

The constructor also determines the category of the metric and hands it as a parameter to the superclass constructor.

The `updateMetricValue` method evaluates the current state of the associated claim and updates the metric accordingly. The method in the example is simple. Other update methods might compute more complex values. You can use the `MetricUpdateHelper` passed to the update method to find out if relevant entities changed. For example:

```
if (helper.updateContainsChangesOfType(History)) {
 // A history event was added, updated or removed
 // You can access the changed items by using
 // Metric.Bundle.getAllModifiedBeansOfType(History).
}
```

9. If you want to apply the new claim metric or indicator to claims that already have metrics, you must enable the `ClaimException` rule **CER04000 Recalculate claim metrics**.

a. Navigate to **configuration** → **Rule Sets** → **Exception** → **ClaimExceptionRules**.

b. Right click **CER04000 Recalculate claim metrics**, and click **Active**. If you see a message asking if you want to edit the file, click **Yes**.

This rule verifies that all claim metrics, exposure metrics, and claim indicators are created on every claim. If there are any missing metrics or indicators in a claim, ClaimCenter adds them to the claim.

10. Save your work in Studio

11. If ClaimCenter is running, stop it. At the command prompt open to `ClaimCenter\bin`, press **CTRL+C**, wait for the prompt, `Terminate batch job (Y/N)?`, and then enter the following commands:

```
y
gwcc dev-stop
```

12. To ensure that your data model changes are correct, regenerate the data dictionary. At the command prompt, enter:

```
gwcc regen-dictionary
```

13. Restart the server. At the command prompt, enter:

```
gwcc dev-start
```

---

**IMPORTANT** Before users start creating new claims or processing of existing claims begins, you must perform the following steps in each environment to which you deploy the new configuration. For example, environments might include a development environment, UAT, and production.

---

14. Log in as an administrator, such as user name **su** with password **gw**.
15. Click the **Administration** tab and then click **Metrics and Thresholds** in the left info bar.
16. Click **Edit** and, on the **Claim Metric Limits** tab, add limits for your new metric type. For more information, see “[Managing Metrics and Thresholds](#)” on page 500 in the *Application Guide*.
17. Run batch processes as follows:
- a. Press **ALT+SHIFT+T** to open the **Server Tools** tab.
  - b. Click **Batch Process Info** in the left Sidebar.
  - c. Run the **Claim Health Calculations** batch process to populate metrics on claims that have never had any metrics. Running this batch process does not add the new metric to claims that already have metrics.
  - d. If you have any metrics that implement **RecalculateMetrics**, set **Recalculate Claim Metrics** batch process to run on a regular basis. You can also run it directly for testing.
  - e. If you enabled the **Recalculate claim metrics** rule previously in step 9, click **Work Queue Info** in the left Sidebar. Then run the **ClaimException** batch process writer to begin processing existing claims and adding the new metric.
- For more information on batch processes, see “[Batch Processes and Work Queues](#)” on page 123 in the *System Administration Guide*.

---

**IMPORTANT** If you retire a metric limit by using Gosu or the database without replacing it with a new limit, new claims will continue to use the retired metric limit.

---



# Configuring Recently Viewed Claims

You can configure recently viewed claim information in the **Claim** tab. You use this tab to either create a new claim, search for a specific claim, or access a recently viewed claim from a list. In the default configuration, the lines of business are configured to show the claim number and the insured's name. However, an exception is the workers' compensation line of business. In the default configuration, this line of business displays the claim number and the claimant's (injured workers) name. This makes sense as a carrier can insure a large-sized employer and have many workers' compensation claims from different employees under that one employer. In another example, one adjuster can be working on multiple claims for one insured in the commercial lines of business. It is even possible that all the open claims for one adjuster could belong to the one insured. Therefore, seeing the insured's name is not as informative as seeing the claimant's name.

This topic explains how to configure the recently viewed claims list that is used in **Claim** tab.

This topic includes:

- “Adding a Loss Date to the Recently Viewed Claim List” on page 563

## Adding a Loss Date to the Recently Viewed Claim List

A carrier might find it useful to also see recently viewed claims (from the **Claim** tab) that include other information, such as the loss date. This topic explains how to add the loss date in the recently viewed claims for the auto loss type using Guidewire Studio.

You are working with the following files:

- `ClaimRecentView.etc`
- `ClaimRecentView.xml`

The current format in ClaimCenter is the claim number and the insured's display name. (As mentioned previously, the format for workers' compensation is claim number and claimant's name.) You see this on the **Claim** tab in the user interface, which is part of the `TabBar.pcf` file.

### To add the loss date

1. In Studio, open the `ClaimRecentView.etx` file and add the loss date column as seen in the following example. Bold text indicates the addition.

```
<?xml version="1.0"?>
<!-- This view entity contains any information needed to display the claims
in the recent claims list displayed under the claim tab. If you want to
change how claims are displayed in this list, then ensure the columns you need
are present in this view entity. Also change the display name for this entity
to display the information you want. -->
<viewEntityExtension xmlns="http://guidewire.com/datamodel" entityName="ClaimRecentView">
<viewEntityColumn name="ClaimNumber" path="ClaimNumber"/>
<viewEntityTypekey name="LossType" path="LossType"/>
<viewEntityName name="InsuredDenorm" path="InsuredDenorm"/>
<viewEntityName name="ClaimantDenorm" path="ClaimantDenorm"/>
<viewEntityColumn name="LossDate" path="LossDate"/>
</viewEntityExtension>
```

2. Save your work.

### To add the new display key

1. Create a new display key. Open the `ClaimSessionState` display key. In Studio, navigate to **configuration** → **Display Keys** → **Java** → `ClaimSessionState`.
  2. Place your mouse over it and right click on it. Select **Add**.
  3. For **Display Key Name** type: `Java.ClaimSessionState.DateLabel`.
  4. For **Default Value** type: `{0} {1} {2}`.
- `ClaimSessionState` now shows the new display key.
5. Save your work.

### To modify the Gosu logic

The following steps explain how to add the `LossDate` variable and modify the Gosu logic.

1. Open the `ClaimRecentView.xml` file.
2. Click **Add**. Under the **Name** column, type `LossDate`.
3. Under the **Entity Path** column, type `ClaimRecentView.LossDate`.
4. Modify the gosu logic (as seen in bolded text) in the following example:

```
uses gw.util.GosuStringUtil

final var DISPLAY_LENGTH = 40;

var contactName : String
if (ClaimLossType == LossType.TC_WC) {
 contactName = Claimant != null ? Claimant : Insured
} else {
 contactName = Insured != null ? Insured : Claimant
}

if (ClaimLossType == "AUTO") {
 return displaykey.Java.ClaimSessionState.DateLabel(
 ClaimNumber,
 LossDate.format("MM/dd/yy"),
 GosuStringUtil.abbreviate(contactName, DISPLAY_LENGTH))
} else {
 return displaykey.Java.ClaimSessionState.Label(
 ClaimNumber,
 GosuStringUtil.abbreviate(contactName, DISPLAY_LENGTH))
}
```

The changes include:

- Extending the display length so that information is not truncated in the tab.
- Adding the Auto loss type so that it affects only commercial and personal auto.
- Adding the loss date format.

5. Save your work and exit Studio.

**To update the application**

The following steps explain how to ensure your changes are reflected in the application.

1. If you are currently running your application, you must shut down the server.
2. Restart the server. This step ensures that the application reflects your data model changes.



# Configuring Incidents

There are several different approaches to creating and editing exposures based on incidents in the user interface.

This topic includes:

- “Implicit Incidents” on page 567
- “Explicit Incidents” on page 568
- “Incidents Data Model” on page 569

## Implicit Incidents

After creating a new exposure, a new incident is also created, and the exposure and incident remain bound together from that point. All exposures require a references to an incident. In cases where the user cannot select an incident, such as the exposure type `GeneralDamage`, the incident is created implicitly.

Use the `New Exposure` and `Exposure Detail` screens to edit a mix of exposure and incident fields. For example, the `description` field (which is part of an incident) displays like a normal exposure field.

There are two exposure screens in the `New Claim` wizard and in the main claim file that can create implicit incidents. For example, `NewExposure.pcf` has the following `Variables` definitions for `Exposure` and `Incident`:

- `Exposure`
  - `initialValue` – `Claim.newExposureWithNoIncident(CoverageType, CoverageSubtype, Coverage)`
    - `name` – `Exposure`
    - `type` – `Exposure`
- `Incident`
  - `initialValue` – `exposure.initializeIncident()`
  - `name` – `Incident`
  - `type` – `Incident`

Similar code is defined for these two variables in `NewClaimWizard_NewExposurePopup.pcf`.

The exposure is initially created without an incident, and the enhancement method `Exposure.initializeIncident` then sets the incident up. In the base configuration, this method, defined in `ExposureUI.gsx`, uses the `Exposure.newIncident` method to create a new, implicit incident for the following exposure types:

- `EmployerLiability`
- `GeneralDamage`
- `LossOfUseDamage`
- `LostWages`
- `PersonalPropertyDamage`
- `WCInjuryDamage`

#### See also

- “Explicit Incidents” on page 568
- “Incidents Data Model” on page 569
- “LOB Typelists and Incidents” on page 479

## Explicit Incidents

For injury, vehicle damage, and property damage exposure types, incidents can be created ahead of time on the **Loss Details** screen. They are explicitly linked with an exposure on the **New Exposure** or **Exposure Detail** pages.

On the ClaimCenter screen, you see a drop down of all suitable incidents. You can also create a new incident. For these exposure types, the `Exposure.initializeIncident` method attempts to pre-fill the incident picker. You can always configure his picker and change the pre-filled value. For information on how this method is used to create implicit incidents, see “Implicit Incidents” on page 567.

To add an incident, use the following logic:

1. Choose the best existing incident of the correct type. An incident is considered better if:
  - It has not already been used for another exposure.
  - It contains a vehicle or property on the policy.
2. If you are unable to choose an incident by using the previous criteria, choose the incident by using display name sort order. If there are no incidents of the appropriate type it is left as `null`, so you must create a new one using the **New...** menu item.

### To Create a New Incident Type

After adding a new exposure type and possibly an incident type, you have to decide whether to use the implicit or explicit incident approach. See “Implicit Incidents” on page 567.

It is best to have a single approach for a single incident type, or the user interface becomes confusing. The only exception to this rule is quick claim configuration, described in the next topic.

### Quick Claim Configuration

You can use some of the quick claim pages to create a claim and one or more exposures all on one page. If the page creates the exposure immediately, use the implicit incident style to initialize the exposure. Otherwise, use two steps to set up the exposure. The first one creates the incident and the second associates it with the exposure. Since anyone who uses a quick claim page is always creating an immediate exposure, use the quick claim configuration to create the exposure and incident together. Edit the contents, but not the association between them, on the quick claim page.

# Incidents Data Model

## Gosu and Incidents

At the domain level, Gosu can work with incident types and their properties. The following topics described some Gosu properties and methods that are exposed on the `Claim`, `Exposure`, and `Incident` entities to make working with incidents easier.

### Claim

`Claim` has an `Incidents` array that contains all the incidents on the claim. `Claim` also provides special arrays for access to incidents of a particular type. These arrays have names of the form `IncidentTypesOnly`. For example, `VehicleIncidentsOnly`, `IncidentsOnly`, and `FixedPropertyIncidentsOnly`.

These arrays are typed—`VehicleIncidentsOnly` has type `VehicleIncident[]`—so you can access all the incidents of a particular type without casts. They do not return any incidents that are subtypes of the named type. For example, `Claim.Incidents` and `Claim.IncidentsOnly` are different arrays. `Claim.Incidents` returns all the incidents on the claim, no matter what their type. `Claim.IncidentsOnly` returns only the incidents that actually have the type `Incident` (not ones which are subtypes of `Incident`). The `Only` arrays are read-only. They do not provide methods for adding or removing incidents.

It is advisable to use the `Claim.IncidentTypesOnly` arrays when dealing with Incidents, since you are usually interested only in incidents of a particular type. If you use `Claim.Incidents`, you see implicit incidents as well as the `ClaimInjuryIncident`. These incidents do not show up in the user interface and do not represent incidents in the real world, but exist to hold data for the Exposure or Claim. These kinds of incidents are filtered out of the `Claim.IncidentTypesOnly` arrays.

### Exposure

`Exposure` has an accessor for each incident type, allowing typesafe access to incident subtypes. For example, `Exposure.VehicleIncident` returns a `VehicleIncident` and can be used to:

```
Exposure.VehicleIncident.DriverRelation = "self"
```

The type safe incident accessor returns `null` if the exposure's incident is not the named type. So `Exposure.VehicleIncident` returns `null` on a `BodilyInjuryDamage` exposure. After you are reading the property, you can use a supertype of the actual incident type. For example, `Exposure.MobilePropertyIncident` can be used to read mobile property incident fields on a `VehicleDamage` exposure, because `MobilePropertyIncident` is a supertype of `VehicleIncident`. But if you set the property, you must use an incident of the exact type:

```
Exposure.ExposureType = "GeneralDamage";
var Incident = new Incident(Exposure);
var VehicleIncident = new VehicleIncident(Exposure);
Exposure.Incident = VehicleIncident; // fine
Exposure.MobilePropertyIncident = VehicleIncident; // throws exception
```

This is because all exposures of a particular type must have incidents of a particular type.

Exposures also provide backwards compatibility for incident properties. Previously, the `description` field was directly on the exposure so you could write:

```
Exposure.Description = "whatever";
```

The `description` actually lives on the exposure's incident, so you would normally have to write:

```
Exposure.Incident.Description = "whatever";
```

However, because of the backwards compatibility properties, you can still access `Exposure.Description`. The aim of keeping the old reference was to reduce the work required to port rules and user interface files that use the old exposure properties. Some caveats:

- The backwards compatibility properties are deprecated.

- The backwards compatibility properties only work if an incident actually exists for the exposure. If a new exposure is created without an incident, setting `Exposure.Description` causes a run time exception because the incident that actually holds the description has not been created yet
- All incident properties, for all incident subtypes are visible at the exposure level. So you can set `Exposure.Vehicle` on a `PropertyDamage` exposure without getting a syntax error. However, this fails at runtime because the underlying incident for a `PropertyDamage` exposure does not have a vehicle field. You can still read `Exposure.Vehicle`, no matter what the exposure type, but it returns `null` if the underlying incident does not have a vehicle field. This is not an issue if you have been careful about only setting appropriate fields on your exposures.

There are also some exposure methods for creating or selecting incidents for an exposure:

```
/**
 * Creates a suitable incident for this exposure.
 * Also sets the incident's claim to be this exposure's claim.
 *
 * @return the new incident
 * @scriptable-all
 */
public Incident newIncident();

/**
 * Looks through the existing incidents on the exposure's claim for the incident that looks
 * to be the best match for this exposure. This incident is pre-filled in the new exposure UI
 * as the initial guess for which incident should be used with this exposure, though the user
 * can always override it.
 *
 * An incident is a better match if it is not already in use by another exposure and if it
 * relates to a vehicle or property on the policy.
 *
 * @return the incident that looks to be the best match for this exposure or null if there
 * are no suitable incidents on the claim.
 * @scriptable-all
 */
public Incident findBestIncidentForNewExposure();
```

These are mainly intended for use by the user interface code when a new exposure is created.

## Incident

Incident has a few methods, which are mainly used in PCF files:

```
/**
 * Is this incident used by at least one exposure?
 * @return true if the incident is used by an exposure, false otherwise.
 * @scriptable-all
 */
public boolean isUsedByExposure();

/**
 * Return all the non-exclusive claim contact roles for this incident
 * @return a list of claim contact role objects, possibly empty but never null
 * @scriptable-ui
 */
public ClaimContactRole[] getNonExclusiveRoles();

/**
 * Return all non exclusive contact roles which are suitable for this incident's type and the
 * given contact. Used in the UI to restrict the user to suitable choices when adding a new
 * contact/role pair to the incident.
 *
 * @param contact a contact, possibly null
 * @return an array of suitable roles, or an empty array if there are none.
 * @scriptable-ui
 */
public ContactRole[] getSuitableNonExclusiveRolesFor(Contact contact);
```

The `isUsedByExposure` method is used to disable the `remove incident` button if an incident is in use. The `getNonExclusiveRoles` and `getSuitableNonExclusiveRolesFor` methods are useful when constructing a list view for adding contacts and roles to an incident (exclusive roles can be handled by a simple picker).

## Coverage

Coverage has an incident-related method, which is used when creating a new exposure, if that exposure has a specific coverage.

```
/**
 * If this coverage relates to a particular vehicle or property then get the associated vehicle or
 * fixed property incident. If there is no such incident then create a new one.
 * If this coverage is not related to a particular vehicle or property,
 * or if it is part of a policy that is not attached to a claim, then return null.
 *
 * @scriptable-all
 */
public Incident findOrCreateIncident();
```

This method is used in the `initializeIncident` library method.

## Entities and Typelists Related to Incidents

### Typelists for Injury Incidents

The `InjuryIncident` subtype is the preferred incident type for all injury-related exposure types (see the `ExposureType` typelist).

Each `InjuryIncident` contains the following fields with the given type:

- `Description` : String
- `GeneralInjuryType` : `InjuryType`
- `DetailedInjuryType` : `DetailedInjuryType`
- `MedicalTreatmentType` : `MedicalTreatmentType`
- `LostWages` : Boolean
- `Impairment` : percentage
- `BodyParts`, an array of `BodyPartsDetails` entities. The `BodyPartsDetails` entity contains the following fields:
  - `PrimaryBodyPart` : `BodyPartType`
  - `DetailedBodyPart` : `DetailedBodyPartType`
  - `CompensabilityDecision` : `CompensabilityDecision`
  - `CompensabilityDecisionDate` : datetime
  - `CompensabilityComment` : String



# Configuring Claim Archiving

*Archiving* is the process of moving a closed claim and associated data from the active ClaimCenter 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.

*This topic includes:*

- “Archiving and the Domain Graph” on page 573
- “Archiving in Guidewire ClaimCenter” on page 575
- “Archiving and Encryption” on page 575
- “Selecting Claims for Archive Eligibility” on page 576
- “Retrieving Archived Objects from the Command Line” on page 577
- “Monitoring Archiving Activity” on page 577
- “Configuring Archiving” on page 578
- “Archiving Plugins” on page 581

**See also**

- “More Information on Archiving” on page 133 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 ClaimCenter 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 Guidewire ClaimCenter, the root entity is the `Claim` object. During the archiving of an instance of the domain graph, ClaimCenter leaves behind a skeleton entity that points to the archived entity. In Guidewire ClaimCenter, this skeleton entity is the `ClaimInfo` object.

- 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 ClaimCenter, the boundary defines the entities that relate to a `Claim` object, such as `Exposure`, `Coverage`, `Matter`, and other similar objects.

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 single claim and all its associated entities.

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, ClaimCenter leaves behind (in the main ClaimCenter database) a skeleton entity instance that 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>This skeleton entity—and <b>only</b> this skeleton entity—must implement the <code>RootInfo</code> delegate. In Guidewire ClaimCenter, this skeleton entity is the <code>ClaimInfo</code> object, the stub object for the <code>Claim</code> object, which is the root object in the ClaimCenter domain graph. You <b>cannot</b> 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 ClaimCenter 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 ClaimCenter 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 the Guidewire data model in general, see “The ClaimCenter Data Model” on page 155.
- For a discussion of delegate objects and how to work with them, see “Delegate Data Objects” on page 167.
- For a discussion of the domain graph, see “The Domain Graph” on page 239.

## The Root Info Entity

As ClaimCenter creates a claim, it also creates a `ClaimInfo` entity instance. The `ClaimInfo` instance is a stub that remains in the active database after ClaimCenter archives the claim. Simple searches occur only in the active database, using `Claim` data to find active claims, and comparable `ClaimInfo` data to find archived claims.

A `ClaimInfo` entity instance retains all links to bulk invoices and claim associations. This permits a retrieved claim to remain connected to these multi-claim entity instances, which are outside the domain graph and therefore always in the ClaimCenter database.

Because it implements the `RootInfo` delegate, the `ClaimInfo` entity includes the `ArchiveState` column, which records the archive state of the claim. The `ArchiveState` value, from the `ArchiveState` typelist, is one of the following:

- `archived`
- `retrieving`
- `null`

A null `ArchiveState` indicates that the claim is in the active ClaimCenter database, either because it has never been archived or because it has been successfully retrieved from the archive.

## Archiving in Guidewire ClaimCenter

Archiving is a multi-step process. At a high level, these steps involve the following:

1. The *Archiving Item Writer* batch process queries the database to select claims that are potentially eligible for archiving. The batch process creates a `WorkItem` in the database for each eligible claim. You can also create an archiving work item by doing any of the following, all of which bypass the batch process and its query criteria:
  - Call `IClaimAPI.scheduleForArchive`.
  - Call `IMaintenanceToolsAPI.scheduleForArchive`.
  - Use the `maintenance_tools` command line tool to retrieve a claim.
2. Individual archive workers pick up—one at a time—any archiving work items created during the first step, running further checks on each identified claim and skipping those that do not pass. These eligibility checks are part of the underlying archiving architecture (meaning that they are set in Java code). For example, internal checks prevent ClaimCenter from archiving a claim with aggregate limits.
3. Finally, if all checks pass, a worker moves the domain graph data from the ClaimCenter database to the archive backing store. ClaimCenter 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.*

### See also

- “Archiving Plugins” on page 581 for a brief description of the archiving plugins.
- “Archiving Integration” on page 575 in the *Integration Guide* for detailed information about the archiving plugins.
- For more information about the criteria used by the query used for claim selection, see “Selecting Claims for Archive Eligibility” on page 576.

## Archiving and Encryption

You are responsible for implementing the `IArchiveSource` plugin interface in such a way as to provide any necessary data encryption. ClaimCenter does not encrypt the values in the XML that it generates and passes to the `IArchiveSource` plugin. However, ClaimCenter 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 Claims for Archive Eligibility

ClaimCenter bases the criteria that determine whether a claim is eligible for archive on the `Claim.DateEligibleForArchive` property. Specifically, for a claim to be archivable, its `DateEligibleForArchive` property must store a non-null date and time *that is not later than the current system date and time*.

In the base ClaimCenter configuration, ClaimCenter manages the value of `Claim.DateEligibleForArchive` from multiple places:

Claim event	ClaimCenter action	DateEligibleForArchive value
Claim created	ClaimCenter does <b>not</b> set the <code>DateEligibleForArchive</code> property as it creates a claim. Therefore, the value of the <code>DateEligibleForArchive</code> property is <code>null</code> .	<code>null</code>
Claim closed	ClaimCenter triggers a Claim Closed rule ( <code>CCL04000 - Set archive eligibility date</code> ) as it closes a claim. This rule sets the value of <code>Claim.DateEligibleForArchive</code> to a date computed by adding the value of the <code>DaysClosedBeforeArchive</code> configuration parameter (in days) to the current system date.	<code>DaysClosedBeforeArchive + current date</code>
Claim retrieved from archive	ClaimCenter uses base configuration class <code>ClaimInfoArchiveSource</code> (which extends <code>ArchiveSource</code> ) to set the value of <code>DateEligibleForArchive</code> . This value is a date computed by adding the value of the <code>DaysRetrievedBeforeArchive</code> configuration parameter (in days) to the current system date.	<code>DaysRetrievedBeforeArchive + current date</code> .
Claim reopened	ClaimCenter triggers a Claim Reopened rule ( <code>CRO04000 - Clear archive eligibility date</code> ) as it reopens a claim. This rule sets <code>DateEligibleForArchive</code> to <code>null</code> .	<code>null</code>

Thus:

- To change the amount of time after a claim has been closed before ClaimCenter archives it, edit the `DaysClosedBeforeArchive` configuration parameter.
- To change the amount of time after a claim has been retrieved from the archive before ClaimCenter archives the claim again, edit the `DaysRetrievedBeforeArchive` configuration parameter.
- To achieve a more fine-grained control over the `DateEligibleForArchive` property, you can edit one of the configuration points listed in the previous table or add code elsewhere to modify it.

**IMPORTANT** If your installation includes claims that you closed before you implemented archiving, then you need to detect those claims and set the `Claim.DateEligibleForArchive` date based on your business requirements. You must also do this if you previously implemented a database-backed version of archiving.

**IMPORTANT** After you implement archiving, you need to set the `DateEligibleForArchive` property on any closed claims that you load through staging tables. ClaimCenter does **not** set a `DateEligibleForArchive` date on these claims.

## Retrieving Archived Objects from the Command Line

An administrator can retrieve one or a group of claims from the command line, by using `maintenance_tools` in `admin/bin`.

To retrieve a single claim, type:

```
maintenance_tools.bat -restore comment -claim claimnumber -user user -password password
```

To retrieve a group of claims, use the same command, but use `file` to name a text file containing a list of claim numbers, separated by new lines.

```
maintenance_tools.bat -restore comment -claim file -user user -password password
```

### See also

- “Maintenance Tools Command” on page 183 in the *System Administration Guide*

## Monitoring Archiving Activity

ClaimCenter 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.  Running the archiving work queue writer is equivalent to running the Archiving Item Writer batch process.
Archive Info	The Server Tools → Info Pages → Archive Info page provides status information about the archive process. It includes information on the following: <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> The Archive Info page provides tools to reset various archive items as well. See “Info Pages” on page 169 in the <i>System Administration Guide</i> for more information.

### Errors in the Archiving Process

If an API or user interface operation attempts to open or work on an archived claim, ClaimCenter typically generates an `EntityStateException`. If an archive worker cannot archive a claim for any reason, it flags the claim as `ExcludedFromArchive`. The Archive Info page `Excluded from Archive` shows the number of excluded claims.

### 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 ClaimCenter 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 Claims” on page 27 in the *System Administration Guide*.

# Configuring Archiving

In working with archiving, you can configure the following:

- Archiving-related Configuration Parameters
- Archive Rules
- Archive Events
- Archive Work Queue

## 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 <code>true</code>) or disabled (set to <code>false</code>). Default is <code>false</code>.</p> <p>This parameter controls the creation of indexes on the <code>ArchivePartition</code> column. If set to <code>true</code>, ClaimCenter creates a non-unique index on that column for <code>Extractable</code> entities. Furthermore, if the <code>Extractable</code> entity is <code>Keyable</code>, ClaimCenter creates a unique index on the <code>ID</code> and <code>ArchivePartition</code> columns.</p> <p><b>WARNING</b> If you set <code>ArchiveEnabled</code> to <code>true</code>, the server refuses to start if you subsequently set it to <code>false</code>.</p> <p>See “<code>ArchiveEnabled</code>” on page 37.</p>
AssignClaimToRetriever	Boolean	<p>Specifies to whom ClaimCenter assigns a retrieved claim:</p> <ul style="list-style-type: none"> <li>• <code>True</code> assigns the claim to the user who retrieved the claim.</li> <li>• <code>False</code> assigns a retrieved claim to the original group and user who owned it.</li> </ul> <p>See “<code>AssignClaimToRetriever</code>” on page 37.</p>
DaysClosedBeforeArchive	Integer	<p>Used by the <code>Claim Closed</code> rule in the base configuration to set the <code>DateEligibleForArchive</code> property on <code>Claim</code>, which determines the date on which ClaimCenter archives a claim automatically.</p> <p>The default in the base configuration is 30.</p> <p>See “<code>DaysClosedBeforeArchive</code>” on page 37.</p>
DaysRetrievedBeforeArchive	Integer	<p>Used by the implementation of the <code>IArchiveSource</code> plugin in the base configuration to set the <code>DateEligibleForArchive</code> property on <code>Claim</code> as it retrieves a claim from the archive store.</p> <p>The default in the base configuration is 100.</p> <p>See “<code>DaysRetrievedBeforeArchive</code>” on page 38.</p>
DomainGraphKnownLinksWithIssues	String	<p>Use to define a comma-separated list of foreign keys from an entity outside of the domain graph that point to an entity inside the domain graph. Naming the foreign key in this configuration parameter suppresses the warning that the domain graph validator would otherwise typically generate for the link.</p> <p>Specify each foreign key on the list as the following:</p> <p style="padding-left: 2em;"><code>relative_entity_name:foreign_key_property_name</code></p> <p>See “<code>DomainGraphKnownLinksWithIssues</code>” on page 53.</p>

Parameter	Type	Description
DomainGraphKnownUnreachableTables	String	<p>Use to define a comma-separated list of relative names of entity types that are linked to the domain graph through a nullable foreign key. This can be problematic as an entity can become unreachable from the graph if the foreign key is null. Naming the type in this configuration parameter suppresses the warning that the domain graph validator would otherwise typically generate for the type.</p> <p>See “<a href="#">DomainGraphKnownUnreachableTables</a>” on page 53.</p>
RestorePattern	String	<p>Code of the activity pattern that ClaimCenter uses to create retrieval activities. Upon retrieving a claim, ClaimCenter creates two activities:</p> <ul style="list-style-type: none"> <li>One activity for the retriever of the claim</li> <li>One activity for the assigned user of the claim, if different from the retriever</li> </ul> <p>The default in the base configuration is <code>restore</code>.</p> <p>See “<a href="#">RestorePattern</a>” on page 38.</p>
SnapshotEncryptionUpgradeChunkSize	Integer	<p>Limits the number of claim snapshots that ClaimCenter upgrades after a change to the encryption plugin or during a change to encrypted fields. Set this parameter to zero to disable the limit.</p> <p>The default in the base configuration is 5000.</p> <p>See “<a href="#">SnapshotEncryptionUpgradeChunkSize</a>” on page 38.</p>

### See Also

- “[Archive Parameters](#)” on page 36
- “[Domain Graph Parameters](#)” on page 53

## Archive Rules

Through the Archive rules, you can do the following:

Skip a claim	<p>Skipping a claim during archiving makes that claim temporarily unavailable for archiving during this particular archiving pass. To skip a claim, call the following method on the claim and provide a reason:</p> <pre>claim.skipFromArchiving(String)</pre> <p><b>IMPORTANT</b> Calling this method on a claim terminates rule set execution.</p>
Exclude a claim	<p>Excluding a claim from archiving makes that claim unavailable for archiving during this and future archiving passes. This makes the claim not archivable on a semi-permanent basis. To exclude a claim from archiving, call the following method on the claim and provide a reason for the exclusion:</p> <pre>claim.reportArchivingProblem(String)</pre> <p>Calling this method on a claim does <b>not</b> terminate rule set execution.</p>

You can view information about skipped and excluded claims in the [Server Tools](#) → [Info Pages](#) → [Archive Info](#) page. This page lists:

- Total number of claims skipped by the archive process
- Number of claims excluded because of rules
- Number of claims excluded because of failure

For skipped and excluded claims, you can investigate each individual item. You can also reset any excluded claim so that the archive process attempts to archive that claim the next time the archive process runs.

### The Default Group Claim Archiving Rule Set

The Archive rule set category contains the Default Group Claim Archiving rule set. ClaimCenter runs the rules in this rule set on each claim in the archive work queue during the archive process. In the base configuration, Guidewire provides the following sample rules in this rule set. Guidewire expects you to customize the archiving rules to meet your business needs.

Rule	Checks...
Claim State Rule	<i>If the claim is closed.</i> If not, then ClaimCenter skips this claim.
Bulk Invoice Item State Rule	<i>If the claim is linked to a Bulk Invoice item with a Draft or Not Valid status, or a status of:</i> <ul style="list-style-type: none"> <li>• Approved</li> <li>• Check Pending Approval</li> <li>• Awaiting Submission</li> </ul> <i>If so, ClaimCenter skips this claim.</i>
Open Activities Rule	<i>If the claim has open activities.</i> If so, then ClaimCenter skips the claim. <b>Note</b> The Work Queue writer does not typically mark a claim with open activities for archiving. An activity can open on the claim between the time the worker queued the claim for archive and the time that the archive batch process actually processes the claim.
Incomplete Review Rule	<i>If there are incomplete reviews on the claim.</i> If so, ClaimCenter skips the claim.
Unsynced Review Rule	<i>If a claim has reviews that have not been synchronized with ContactManager.</i> If so, ClaimCenter skips the claim.
Transaction State Rule	<i>If a claim has transactions that have yet to be escalated or acknowledged.</i> If so, ClaimCenter skips the claim.

It is possible for you to add your own, additional, archive-related rules to this rule set, or to modify the sample rules that Guidewire provides. Thus, it is possible to write archiving rules that reflect your unique business conditions. For example, you can write rules to do the following:

- Do not archive a sensitive claim, or one with restricted access control.
- Do not archive a claim that meets a certain condition, such as having medical payments over some amount.
- Do not archive a claim whose claimant has other open claims pending.

There are additional archiving-related methods, such as `Claim.hasReportedArchiveProblem`, that are useful in rule writing.

#### See also

- For more information on archiving rules, see “Archive Rule Set Category” on page 40 in the *Rules Guide*.

## Archive Events

See “Detecting Archive Events” on page 41 in the *Rules Guide* for information on the events associated with archiving.

## Archive Work Queue

The Archive batch process writes items to the Archive work queue. Items in this queue correspond to possible archivable items. As ClaimCenter processes these items, it writes the associated claims to XML.

In working with archiving:

- You can modify the rules that control which claims the Archive work queue archives. See “Archive Rule Set Category” on page 40 in the *Rules Guide*. See also “Archive Rules” on page 579.
- You can configure how the Archive writer and worker daemons behave. See “Distributable Work Queues” on page 124 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 577 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 44 in the *System Administration Guide*.
- “Distributable Work Queues” on page 124 in the *System Administration Guide*
- “Configuring Distributable Work Queues” on page 128 in the *System Administration Guide*

## Archiving Plugins

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.archiving.ClaimInfoArchiveSource`
- 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 ClaimCenter *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 575 in the *Integration Guide*



# Configuring Special Instructions

Special handling instructions are a set of additional steps in claims processing that can be associated with accounts or service tiers.

This topic explains how to configure special instructions, specifically service tiers, email notifications, and claim headline comments.

The following configuration tasks can be performed:

- Create and enable new service tiers.
- Configure contact roles for special handling notifications.
- Configure Instruction Types and Instruction Categories for claim headline comments.

**See also**

- “Accounts and Service Tiers” on page 109 in the *Application Guide*.

This topic includes:

- “Creating Service Tiers” on page 583
- “Configuring Email Notifications” on page 585
- “Configuring Claim Headline Comments” on page 585
- “Configuring Activity Patterns for Special Handling” on page 586

## Creating Service Tiers

Service tiers represent level of customer service to be provided to a group of policies. Service tier information is typically included in both the policy administration system (PAS) and ClaimCenter. In the base configuration, the `ServiceTier` attribute is on the `Account` entity in PolicyCenter, and the `CustomerServiceTier` attribute is on the `Policy` entity in ClaimCenter.

ClaimCenter provides three service tiers as samples in the base configuration:

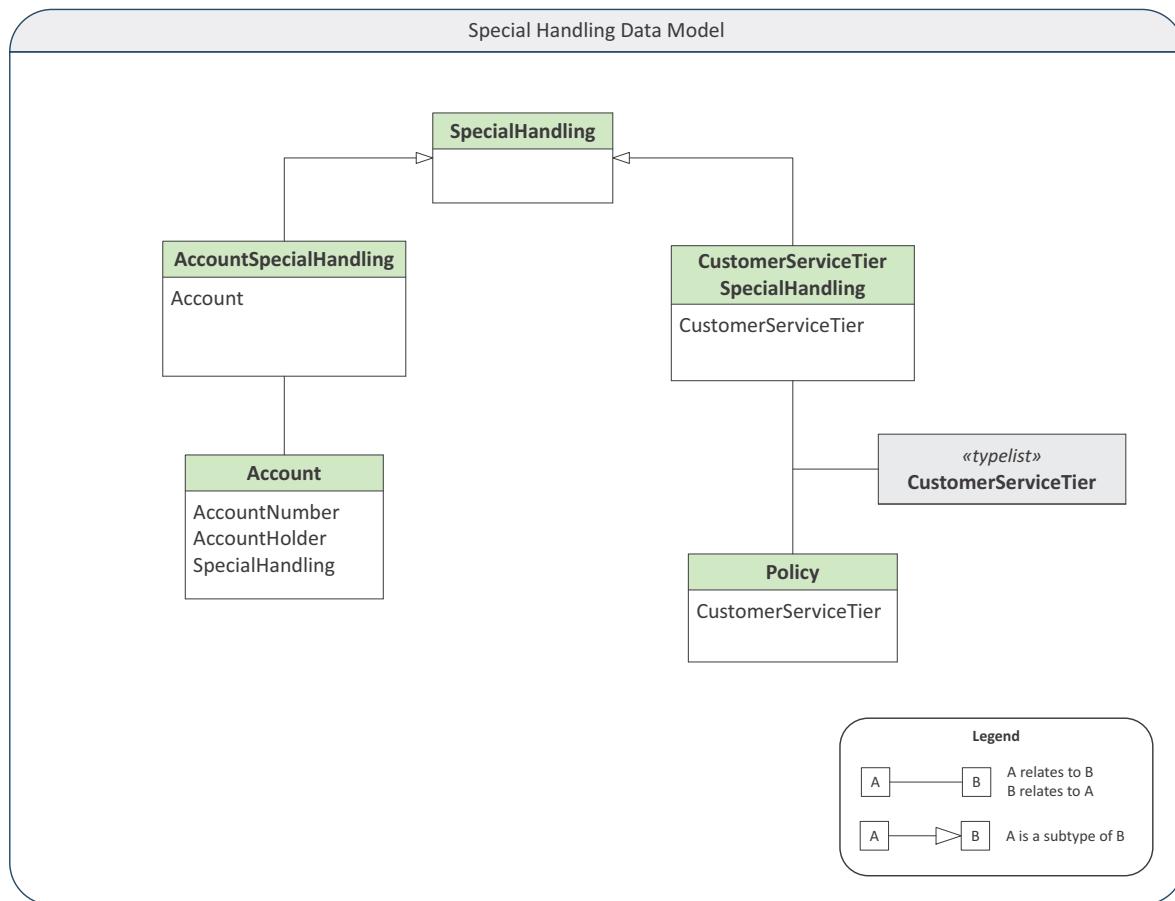
- Platinum
- Gold

- Silver

The Platinum and Gold tiers are active, and the Silver is not enabled. These are provided merely as examples and have no viable functionality associated with them.

## Special Handling Data Model

Service tiers are represented by the `CustomerServiceTier` typelist, which is an attribute on the `Policy` entity. The following diagram illustrates the relationships in the `SpecialHandling` data model.



## Adding Service Tiers

You can add service tiers in ClaimCenter using a two-step process. Service tiers in ClaimCenter must be mapped to the corresponding fields in the PAS for them to be utilized appropriately. Discuss with your integration developer before creating new service tiers.

1. Start ClaimCenter Studio.

2. In `configuration` → `config` → `Extensions` → `Typelist`, double-click and open the `CustomerServiceTier.ttx` typelist.

3. Right-click and select **Add new...** → **typecode** and enter the values for your new service tier. For example, enter the following values:

Code	Name	Description	Priority	Retired
Bronze	Bronze Customer	The service for Bronze customers.	-1	false

**Note:** Restart the server before proceeding to enable the service tier.

4. In the **Administration** menu, click **Administration** → **Special Handling** → **Service Tiers** and select **Add Service Tier** to add the tier. See “Adding Service Tiers” on page 112 in the *Application Guide* for more details.

## Configuring Email Notifications

Special handling instructions include automated email notifications that can be triggered based on claim indicator or financial events. See “Working with Automated Notifications” on page 114 in the *Application Guide*.

Email notifications can be sent to single or multiple recipients, or contacts with roles on the claim.

You can configure the claim contact roles that are available for email notifications using the steps below:

1. Start ClaimCenter Studio.
2. In **configuration** → **config** → **Extensions** → **Typelist**, double-click and open the **ContactRole.ttx** typelist.
3. Select the **SpecialHandling** typefilter.
4. Add the contact role to the typefilter, as follows:
  1. Right-click, select **Add new...** and click **Include Into Filter....**
  2. Select the desired claim contact role and click **Finish**.The role now appears in the **SpecialHandling** typefilter list.
5. Restart the application server. You can now see the recently added contact role in the **Contact Role** field of the **New Automated Notification** screen.

## Configuring Claim Headline Comments

Claim headline comments are note-like comments shown in the claim headline and are created under the **Other Instructions** section of the Special Handling menu. Claim headline comments can be created based on policy types, instruction categories and instruction type.

You can configure the instruction categories and types available in the **Other Instructions** screen. You will need to create extensions for the **InstructionCategory** and **InstructionType** typelists in Studio, as shown below:

1. Start ClaimCenter Studio.
2. In **configuration** → **config** → **Metadata** → **Typelist**, right-click the **InstructionCategory.tti** typelist and select **New Typelist Extension**.
3. In the **Typelist Extension** window, click **OK** to accept the default name, **InstructionCategory.ttx**.
4. The typelist extension file is opened. In **InstructionCategory.ttx**, right-click **InstructionCategory** and select **Add new... → typecode**.
5. Enter the code, name, and description. Select **Save**.
6. Repeat Step 2 and Step 3 to extend the **InstructionType.tti** typelist.

7. In the resulting **InstructionType.ttx** typelist, right-click to add either a new typecode or a category to one of the existing typecodes. Click **Save**.
8. Restart the application server.

## Configuring Activity Patterns for Special Handling

Special handling includes the creation of automated activities, based on predefined claim indicator or financial events.

An activity pattern must specifically be configured for special handling, as shown below.

1. Navigate to **Administration** → **Business Settings** → **Activity Patterns**.
2. Select **Add Activity Pattern** to create a new activity pattern, or select an existing activity pattern and click **Edit**.
3. In the **Category** field, select **Handling Instructions**.

The activity pattern is now marked for special handling and will be available for selection in the **Special Handling** menu.

# Configuring Roles and Relationships

This topic describes how to customize and configure contact roles and relationships in a Guidewire application. It supplies some common examples you can use to configure your own installation. The roles discussed in this topic apply only to contacts. Another type of role is the user role, which is a collection of permissions that enable a user to view or edit various ClaimCenter or ContactManager objects.

This topic includes:

- “Adding Contact Roles” on page 587
- “How Configuring Roles Impacts Entity Data and Types” on page 591
- “Adding a New Contact Role: an Example” on page 594
- “Relationships Between Contacts” on page 596
- “Adding a Bidirectional Contact Relationship: an Example” on page 597

**See also**

- For information on user roles and security related to accessing contacts in the Address Book, see “Securing Access to Contact Information” on page 109 in the *Contact Management Guide*.
- For information on roles in general and on roles used in ClaimCenter, see “Understanding Roles” on page 439 in the *Application Guide*.

## Adding Contact Roles

The contacts in the system have associations with your business entities that depend on the functions they perform for the business entities. For example, a Person can be associated with a claim as the insured party. In ClaimCenter, these associations are defined as roles that contacts perform for claims, evaluations, incidents, exposures, negotiations, matters, and policies. For example, a defense attorney performs a role in a matter that involves litigation on a claim.

The ClaimCenter base configuration has a number of predefined contact roles. Your business might require unique associations that would necessitate adding your own roles.

**Note:** Do not add a contact role to a custom entity. ClaimCenter does not support this configuration.

To add a contact role, you define an association between a role and a contact type, and then associate that role to an entity, as follows:

1. Add the role to the `ContactRole` typelist as described in “Defining Contact Roles” on page 588.
2. Define the following two types of constraints for the role in the `entityroleconstraints-config.xml` file, as described in “Defining Role Constraints” on page 588:
  - A *contact role constraint* indicates the contact subtype, such as `Person` or `Company`, to which the role applies. See “Defining Contact Role Constraints” on page 589.
  - An *entity role constraint* defines which roles an entity can use. It can also optionally specify whether a role is required, exclusive, or prohibited, and conditions that apply to use of the role by the entity. See “Defining Entity Role Constraints” on page 589.
3. After completing work on the typelist and the XML file, you must rebuild and redeploy ClaimCenter. Guidewire also recommends that you regenerate the *Data Dictionary*. In any case, Guidewire requires that you regenerate the *Data Dictionary* for production deployments.

## Defining Contact Roles

The `ContactRole` typelist defines the available roles. For example, the following contact role definition creates a `mattermanager` role:

Code	Name	Description
<code>mattermanager</code>	Legal Case Manager	Legal Case Manager

Many of the codes in the `ContactRole` typelist, such as `activityowner`, `claimant`, `insured`, `vendor`, and `venue`, are internal to the application. The typelist editor lists the internal codes with a `Code` field that has a gray background. The editor does not let you remove these typecodes.

You can also look up the `ContactRole` typelist in the *Data Dictionary*. The dictionary marks the `Internal` column `true` for the codes that Guidewire defines as internal.

**IMPORTANT** In general, if you remove any typecodes from any typelist, first ensure that the code is not in use in the application instance in a PCF file. Also verify that another typelist does not reference that typecode.

Contact roles exist in relation to entities. You define which entities use a role by referencing the role from the `entityroleconstraints-config.xml` file. Topic “Defining Role Constraints” on page 588 describes this process.

### To view or edit a typelist

1. Open Guidewire Studio, if it is not already running. To do so, at a command prompt, navigate to `ClaimCenter/bin` and enter the following command:  
`gwcc studio`
2. In the **Resources** pane on the left, expand **Typelists** and click **ContactRole**.

## Defining Role Constraints

You configure the relationships between entities and roles in the `entityroleconstraints-config.xml` file. In this file, you define which `Contact` subtypes can use a role. Additionally, you configure which contact roles are available to which entities, such as a `Claim` or an `Exposure` entity. As described previously in “Defining Contact Roles” on page 588, you must define the contact role codes that you use in

entityroleconstraints-config.xml in the ContactRole typelist.

**Note:** Only the entities supplied in the base configuration and configured on ClaimContactRole can use roles. These entities are Claim, Evaluation, Exposure, Incident, Matter, Negotiation, and Policy.

To view or edit the file, in ClaimCenter Studio, expand Other Resources and click entityroleconstraints-config.xml in the Resources pane on the left.

## Defining Contact Role Constraints

The ContactRoleTypeConstraint element defines the Contact subtype to which the role belongs. For example, a mattermanager is of type Person, as the following code defines it:

```
<ContactRoleTypeConstraint contactRoleCode="mattermanager" contactSubtype="Person"/>
```

In this example:

- The contactRoleCode specifies the role's code name, which is mattermanager.
- The contactSubtype identifies the subtype to which the role belongs, which is Person.

If each of your contact roles uses only one Contact subtype, ClaimCenter can ensure that the role is assigned to the correct contact type. Additionally, your PCF configuration is relatively simple.

However, if a role uses more than one subtype, then you must perform extra configuration. This is to ensure that selection of this role does not result in assignment of the wrong contact type. Even with extra configuration, it is possible to have the wrong type assigned to the role, as explained in the following claimant role description.

The base configuration roles are all set up to use one subtype, with a single exception, the claimant role, which can be both a Person and a Company. To specify the additional subtype, this configuration uses the ExceptionConstraint tag, as shown in the following example:

```
<ContactRoleTypeConstraint contactRoleCode="claimant" contactSubtype="Person">
 <ExceptionConstraint contactSubtype="Company" entityRef="Exposure"/>
</ContactRoleTypeConstraint>
```

The ExceptionConstraint tag ensures that the claimant role is always performed by a Person except when using an Exposure entity, when a Company can also perform this role. As the system starts up, it calculates the nearest supertype of the two subtypes and effectively assigns the claimant role to that supertype. For the claimant role, the nearest supertype is Contact.

**Note:** Calculating the nearest supertype can result in assignment of an incorrect contact type unless you account for this possibility in your configuration. The possibility of incorrect type assignment is one reason Guidewire encourages you to associate a contact role with a single subtype.

## Defining Entity Role Constraints

The EntityRoleConstraint block defines the associations between roles and entities. The following example from the base configuration entityroleconstraints-config.xml file shows the uses of the key configuration tags:

```
<EntityRoleConstraint>
 <EntityRef entityType="Claim">
 <RoleRef contactRoleCode="FirstIntakeDoctor">
 <RoleConstraint constraintType="Exclusive"/>
 </RoleRef>
 <RoleRef contactRoleCode="InsuredRep"/>
 <RoleRef contactRoleCode="LawEnfcAgcy"/>
 <RoleRef contactRoleCode="OccTherapist"/>
 <RoleRef contactRoleCode="PhysTherapist"/>
 <RoleRef contactRoleCode="PrimaryDoctor">
 <RoleConstraint constraintType="Exclusive"/>
 </RoleRef>
 <!-- some definitions skipped for brevity -->
 <RoleRef contactRoleCode="claimant">
 <RoleConstraint constraintType="Exclusive"/>
 <RoleConstraint constraintType="Required">
 <AdditionalInfo propertyName="LossType" value="WC"/>
 </RoleConstraint>
 </RoleRef>
</EntityRef>
</EntityRoleConstraint>
```

```

<RoleConstraint constraintType="Prohibited">
 <AdditionalInfo propertyName="LossType" value="AUTO"/>
 <AdditionalInfo propertyName="LossType" value="PR"/>
 <AdditionalInfo propertyName="LossType" value="GL"/>
</RoleConstraint>
</RoleRef>
<RoleRef contactRoleCode="claimantdep"/>
<RoleRef contactRoleCode="codefendant"/>
...
</EntityRef>
</EntityRoleConstraint>

```

An `EntityRef` tag designates an `entityType`, an entity that can use contact roles, and contains one or more `RoleRef` tags that define contact roles for the entity. By default, there are no constraints on the entity's use of the role. In the previous example, a `Claim` can use or not use the `InsuredRep`, `LawEnfcAgcy`, `OccTherapist`, `PhysTherapist`, `claimantdep`, and `codefendant` roles without restriction.

### Adding Constraints to a Contact Role

You can constrain the relationship between an entity and a role by putting a `RoleConstraint` tag in the `RoleRef` block. As shown in the previous `<EntityRoleConstraint>` example, the `FirstIntakeDoctor`, `PrimaryDoctor`, and `claimant` contact role codes use this tag. See “`<EntityRoleConstraint>`” on page 589. The following `constraintType` values are possible:

<b>Exclusive</b>	At most one contact associated with the entity can fulfill this role. It is possible that the entity does not have anyone in this role.
<b>Required</b>	This entity must have at least one contact in this role.
<b>Prohibited</b>	None of the contacts for this entity can have this role. Often qualified with an <code>AddtionalInfo</code> tag to limit the application of this constraint.
<b>ZeroToMany</b>	This entity can have no contact, one contact, or many contacts that use this role, which is the default behavior for a role that has no constraint types defined. Do not use this <code>constraintType</code> with <code>Exclusive</code> .

If you want to ensure that an entity has exactly one contact that fills a particular role, add two `RoleConstraint` tags to the `RoleRef`—an `Exclusive` and a `Required` constraint.

In the previous `<EntityRoleConstraint>` example, the `FirstIntakeDoctor` role has an `Exclusive` constraint. This constraint means that a `FirstIntakeDoctor` is not required for a `Claim`, but if a `FirstIntakeDoctor` is needed for the `Claim`, only one can be assigned. The same is true for a `PrimaryDoctor`.

### Putting Limitations on Constraints

A constraint can be further refined with `AdditionalInfo` tags. An `AdditonalInfo` tag specifies a `propertyName` and `value` that must exist for the system to apply the constraint. In the previous `<EntityRoleConstraint>` example, a `claimant` has three constraints:

- `Exclusive`
- `Required`
- `Prohibited`.

See “`<EntityRoleConstraint>`” on page 589.

The `Exclusive` constraint applies to all allowable types of claims, limiting the number of claimants on these claims to no more than one. The `Required` and `Prohibited` constraints have `AdditionalInfo` tags that restrict their application. If the `LossType` for a claim is `WC` (workers' comp), then the `claimant` role is `Required` and must

be filled. If the `LossType` for a claim is `AUTO`, `PR` (property), or `GL` (general liability), the `claimant` role is `Prohibited` and cannot be filled at all.

**Note:** For auto, property, and general liability claims, the claimants do not exist at the claim level. For these types of claims, you cannot add a claimant until you have exposures. The `claimant` contacts are owned by the exposures, not by the claim itself, because each exposure can have a separate claimant. For workers' comp claims, there is always a single claim, regardless of the number of exposures. Therefore, a workers' comp claim can (and must) own a contact with the role of `claimant`.

### Resolving Conflicts in Role Constraint Configuration

It is possible to specify role constraints that can result in conflicts between constraints. For example, take the following fictional configuration:

```
<EntityRef entityType="Claim">
 <RoleRef roleCode="claimant">
 <RoleConstraint constraintType="Required">
 <AdditionalInfo propertyName="LossType" value="AUTO"/>
 </RoleConstraint>
 <RoleConstraint constraintType="Prohibited">
 <AdditionalInfo propertyName="State" value="CA"/>
 </RoleConstraint>
 </RoleRef>
</EntityRef>
```

If, at run time, a `Claim` has a `LossType` of `AUTO` and a `State` value of `CA`, there is a conflict. Auto claims require a `claimant`, but according to this constraint definition, in California, a `claimant` is prohibited on a claim.

ClaimCenter resolves this conflict by calculating constraint precedence and using the constraint with the highest precedence. The following list shows the order of constraint precedence from highest to lowest:

- `Prohibited`
- `Exclusive & Required`
- `Exclusive`
- `Required`
- `ZeroToMany`

In the previous example, `Prohibited` has higher precedence than `Required`, so the `claimant` role cannot be assigned on this auto insurance claim in California.

#### See also

- “How Configuring Roles Impacts Entity Data and Types” on page 591
- “Adding a New Contact Role: an Example” on page 594
- “Relationships Between Contacts” on page 596

## How Configuring Roles Impacts Entity Data and Types

Creating or changing a role configuration affects the data definition of the associated entity. When the system generates the entity's type information, it must include information for the role. Depending on how you configure a contact role, the application generates either a simple property or an array property on the entity. The system also provides methods on the entity for manipulating roles in general and for specific properties.

**Note:** If a role is prohibited for an entity, the system does not generate a property or any methods for the role.

## Generated Role Methods

ClaimCenter generates a number of methods on the entities related to a role. You use these methods to manipulate and gather information about roles and contacts. The following table lists some of the methods available on the `Claim` entity:

Method	Description
<code>getClaimant(): Contact</code>	Returns the claimant for a claim or, in the case of an exposure that has a claimant, for the exposure.
<code>getClaimants(): Contact[]</code>	Returns all claimants for a claim, such as an Auto claim that has multiple exposures.
<code>getClaimantNames(): String[]</code>	Returns names of all claimants on a claim as well as those on related exposures.
<code>getClaimContactsForAllRoles(): ClaimContact[]</code>	Gets all <code>ClaimContact</code> entities for all roles.
<code>getContactByRole(role: ContactRole): Contact</code>	Gets the contact serving in an exclusive role. If you call this method on a role that is not exclusive, it throws an exception.
<code>getContactsByRole(role: ContactRole): Contact[]</code>	Gets the directly-related <code>Contact</code> objects in the given role. This method returns only contacts attached directly to the entity. It does not return contacts that are attached to the entity's subobjects.
<code>getContactsByRoles(roles: ContactRole[]): Contact[]</code>	Gets the directly related <code>Contact</code> objects with at least one of the given roles. It does not return contacts related to subobjects.
<code>getContactsExcludeRole(role: ContactRole): Contact[]</code>	Gets the directly related <code>Contact</code> objects that are not in the given role. This method returns only contacts attached directly to the entity. It does not return contacts attached to the entity's subobjects.
<code>getContactsExcludeRoles(roles: ContactRole[]): Contact[]</code>	Gets directly related <code>Contact</code> objects that are not in any of the given roles. It does not return <code>Contact</code> objects related to subobjects.
<code>getContactType(role: ContactRole): IType</code>	Gets the type of the specified contact.
<code>setContactByRole(role: ContactRole, contact : Contact): void</code>	Sets the contact for a specified role if the role is Exclusive or both Exclusive and Required.
<code>addRole(role: ContactRole, contact: Contact): ClaimContactRole</code>	Adds a role with the specified contact to the entity. Use this method only with the Required or ZeroToMany roles. For Exclusive roles or for Exclusive & Required roles, use the explicit <code>setContactByRole</code> method for the role. If either the role or contact does not exist, this method does nothing.
<code>removeRole(claimContactRole: ClaimContactRole): void</code>	Removes a role from the entity. If the role is the only role for the associated contact, it attempts to remove the contact as well. If either the role or contact do not exist on the entity, this method does nothing.
<code>removeRole(role: ContactRole, contact: Contact): void</code>	Removes a role from the entity for this specific contact. Use this method only with Required or ZeroToMany roles. For Exclusive roles or the Exclusive & Required roles, use the other <code>removeRole</code> method. If either the role or contact does not exist on the entity, this method does nothing.

Similar methods are also available on the entities **Exposure**, **Incident**, and **Matter**.

**Note:** In addition to the get methods for **Contact** entities, there are get methods for **ClaimContact** entities that are similar, except that they return **ClaimContact** entities rather than **Contact** entities. To see all the methods associated with these entities, you can use the Gosu API reference. See “Gosu Generated Documentation (“gosudoc”)” on page 37 in the *Gosu Reference Guide*. You can also run the Gosu Tester, which is available in Studio on the **Tools** menu. In the Gosu Tester, you can declare variables of the various entity types and use code completion pop-ups (CTRL+SPACEBAR) to show you what is available for an entity.

## Entity Property for Exclusive or Exclusive & Required Role

If you add an **Exclusive** role or an **Exclusive & Required** role to an entity, the system adds a single property to the entity for the role. You can view this property in the *Data Dictionary*. For example, the **ContactRole** typelist defines the **maincontact** role. File **entityroleconstraints-config.xml** configures this role as follows:

```
...
<ContactRoleTypeConstraint contactRoleCode="maincontact" contactSubtype="Person"/>
...
<EntityRef entityType="Claim">
...
 <RoleRef contactRoleCode="maincontact">
 <RoleConstraint constraintType="Exclusive"/>
 </RoleRef>
</EntityRef>
...
```

When you regenerate the *Data Dictionary*, you see a **maincontact** property on the **Claim** that has a foreign key to the **Person** contact type.

**maincontact** Derived property returning [Person](#) (virtual property)<sup>?</sup> (writable)<sup>?</sup>  
The Main Contact for this Claim

The system also generates getter and setter methods on the **Claim** plugin class for this property—in this example, the **getMaincontact** and **setMaincontact** methods. You can also get and set this property using Gosu, for example, by adding Gosu code to a rule.

## Entity Array Key for Required or ZeroToMany Role

If you add a **Required** or **ZeroToMany** role to an entity, the system adds an array key to the entity. If there is no role constraint specified, the default is **ZeroToMany**. For example, Guidewire configures the **arbitrator** role in the base configuration as follows:

```
...
<ContactRoleTypeConstraint contactRoleCode="arbitrator" contactSubtype="Adjudicator"/>
...
<EntityRef entityType="Claim">
...
 <RoleRef contactRoleCode="arbitrationvenue"/>
 <RoleRef contactRoleCode="arbitrator"/>
 <RoleRef contactRoleCode="assessor"/>
</EntityRef>
...
```

The system generates the array key **arbitrator** on the **Claim** entity. The array key points to an array of **Adjudicator** contacts.

The system also generates a getter on the entity that enables you to get a list of the contacts in the **arbitrator** role. The system does not generate a setter for array properties. To manipulate the array’s contents, use the **addRole** and **removeRole** methods.

## Avoiding Errors with Contact Properties

For every subtype you create, your Guidewire application creates a property for that subtype on its parent. For example, if you extend CompanyVendor with a Dentist subtype, the system creates a Dentist property. In this case, do not create additional database entities with the name Dentist. For example, if you create a custom relationship, do not name the relationship accessor Dentist. If you do use the property name for the name of a database entity, you will receive an error similar to the following:

```
[java] java.lang.ClassCastException: com.guidewire.commons.metadata.internal.loader.ArrayData
[java] at com.guidewire.tools.datadictionary.support.TableWriter.writeColumns(TableWriter.java:239)
[java] at com.guidewire.tools.datadictionary.support.TableWriter.writeSubtype(TableWriter.java:504)
[java] at com.guidewire.tools.datadictionary.support.TableWriter.writeSubtypes(TableWriter.java:476)
[java] at com.guidewire.tools.datadictionary.support.TableWriter.writeTable(TableWriter.java:155)
[java] at com.guidewire.tools.datadictionary.support.DictionaryWriter.writeTable(DictionaryWriter.java:409)
[java] at com.guidewire.tools.datadictionary.support.DictionaryWriter.outputDictionary(DictionaryWriter.java:360)
[java] at com.guidewire.tools.datadictionary.Main.main(Main.java:132)
[java] Exception in thread "main"
```

### See also

- “Adding Contact Roles” on page 587
- “Adding a New Contact Role: an Example” on page 594
- “Relationships Between Contacts” on page 596

## Adding a New Contact Role: an Example

The following example illustrates how to add a new contact role in Guidewire ClaimCenter and how to use that role within ClaimCenter.

1. If necessary, start ClaimCenter Studio.
2. In the Resources pane on the left, expand Typelists and click **ContactRole**.
3. Click **Add** and add a role with the following values:

Code	Name	Description
negotiator	Negotiator	Person who handles a negotiation

Adding this line causes the database upgrader to add a negotiator value to the **ContactRole** typelist.

4. In the Resources pane under Typelists, click the **ContactRoleCategory** typelist to open it in the editor.
5. Select the Vendors category and, below on the Categories tab, click **Add** to add the following **ContactRole**:

TypeList	Code
ContactRole	negotiator

This step sets up filtering by this particular role for parties involved.

6. In the **entityroleconstraints-config.xml** file, add the negotiator role, and then add negotiator as a role reference to the list of Negotiation roles.
  - a. In the Resources pane, expand Other Resources and click **entityroleconstraints-config.xml** to edit it.
  - b. Add the following **ContactRoleTypeConstraint** to the section for role constraints at the top of the file:
 

```
<ContactRoleTypeConstraint contactRoleCode="negotiator" contactSubtype="Person"/>
```

- c. Add the negotiator role with the Exclusive (zero or one interpreter) constraint to the Negotiation entity, <EntityRef entityType="Negotiation">:

```
<RoleRef contactRoleCode="negotiator">
 <RoleConstraint constraintType="Exclusive"/>
</RoleRef>
```

7. Save the file.

8. Stop ClaimCenter, regenerate the APIs, optionally regenerate the data dictionary, and restart ClaimCenter, as described in the steps that follow.

**Note:** Because the role changes and additions are data model changes, you must at least stop and restart ClaimCenter.

- a. If necessary, stop ClaimCenter.

- b. Regenerate the APIs:

```
gwcc regen-java-api
gwcc regen-soap-api
```

**Note:** The regen-soap-api command regenerates RPC-Encoded SOAP APIs only. To regenerate WS-I APIs, stop and restart the application. ClaimCenter regenerates the APIs automatically.

- c. Optionally, regenerate the data dictionary:

```
gwcc regen-dictionary
```

- d. Start the ClaimCenter application.

9. You also need to quit ClaimCenter Studio and restart it to enable use of the Negotiation.negotiator virtual property.

10. In the Studio Resources pane, open the Localizations node.

11. Click Display Keys and navigate to NVV → Matter → SubView → MatterNegotiationDetail → General.

12. Right-click General and click Add, and then add a display key using the following information:

Display key name	NVV.Matter.SubView.MatterNegotiationDetail.General.Negotiator
Default value	Negotiator

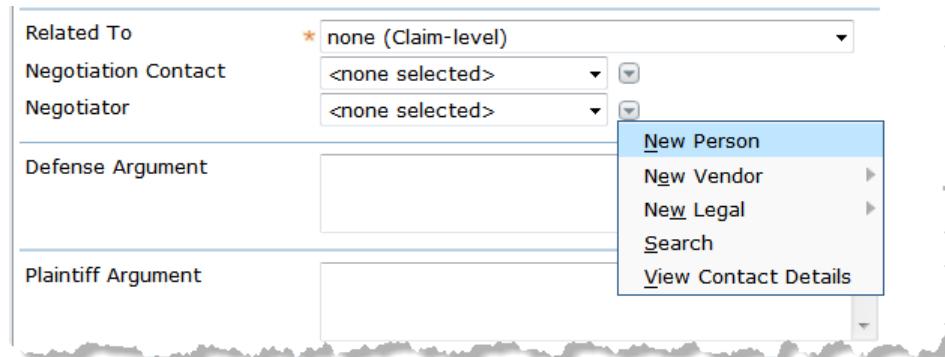
13. In the Resources pane on the left, navigate to Page Configuration (PCF) → claim → newother and click NewNegotiationDV to open this PCF file in the editor.

14. Drag a ClaimContactInput widget from the Toolbox on the right and drop it after the ClaimContactInput with id General\_NegContact, and then select it and set the following properties:

editable	true
id	NegotiatorContact
label	displaykey.NVV.Matter.SubView.MatterNegotiationDetail.General.Negotiator
required	false
value	Negotiation.negotiator
claim	Negotiation.Claim
valueRange	Negotiation.Claim.RelatedContacts as Person[]

Because the Negotiation.negotiator value is a Person, but the valueRange for this field is all Claim contacts, you must cast the valueRange to be an array of Person objects. If you previously regenerated the Data Dictionary, you can use it to view the Negotiation entity and the new negotiator field.

15. Navigate to Page Configuration (PCF) → claim → planofaction and open ClaimNegotiationDetailsDV. Repeat the actions of the previous step. In other words, add a ClaimContactInput widget after the ClaimContactInput with id General\_NegContact and set its properties the same as you did in the previous step.
16. If you are running ClaimCenter in development mode, login as an administrator and click ALT+SHIFT+L to reload your user interface changes. An alternative way to reload a PCF change is to stop and restart ClaimCenter, and then log in again.
17. Open a claim, and then choose Actions → New → Negotiation.
18. When you view the Negotiator field in the New Negotiation window, the system automatically generates a picker for it:



19. Add a negotiator and enter some information about the negotiation and then save the new negotiation.
20. Click Plan of Action on the left, and then click Negotiations in the blue bar at the top to show the current negotiations.
21. Select the negotiation you created and look for the Negotiator field, which shows the negotiator you added. You can click Edit to change the settings for the negotiation, including the Negotiator.

#### See also

- “Adding Contact Roles” on page 587
- “How Configuring Roles Impacts Entity Data and Types” on page 591
- “Relationships Between Contacts” on page 596

## Relationships Between Contacts

Relationships enable you to associate two contacts by applying an association to them. Examples of associations are employer and employee or attorney and law firm. You see a contact’s relationships on the ClaimCenter Related Contacts subtab.

For example, if you open a claim in ClaimCenter, then click Parties Involved on the left, you see a list of contacts for the claim. Below this list is an Edit window for the selected contact that has three tabs, one of which is Related Contacts.

Relationships are bidirectional automatically. If you add a Person contact to a Company contact as an Employee, the company appears on the person’s Related Contacts tab as an Employer. Additionally, the person appears on the company’s Related Contacts tab as an Employee.

You configure contact relationships within ClaimCenter by doing the following:

- Editing file contact-relationship-config.xml to add a contact relationship pair
- Modifying the ContactBidiRel and ContactRel typelists to add the new contact relationships

- Modifying the `RelationshipSyncConfig` class to ensure that your synchronization configuration also has the primary relationship defined as a synchronization attribute

In the `contact-relationship-config.xml` file there are a number of internal contact relationships: `guardian`, `employer`, `primarycontact`, and so on. To add your own relationships, add them to the end of the file.

**IMPORTANT** The system relies on these internal relationships. Do not modify or remove the internal contact relationships. However, you can use a filter attribute to filter these values in a PCF drop-down list.

## Adding a Bidirectional Contact Relationship: an Example

This example shows how to add a relationship between two contacts to indicate the primary and secondary insured parties on a claim.

**Note:** This example requires that you first integrate Guidewire ContactManager with Guidewire ClaimCenter. See “ContactManager Integration Reference” on page 269 in the *Contact Management Guide* for details on how to integrate the two applications.

### To add a new relationship to ClaimCenter

1. If necessary, start ClaimCenter Studio.

At a command prompt, navigate to the ClaimCenter `bin` directory and enter the following command:  
`gwcc studio`

2. In the Project window, navigate to `configuration → config → Metadata → Typelist`.
3. Right-click `ContactRel.tti` and choose `New → Typelist Extension` and then click `OK`.  
Studio creates `ContactRel.ttx` in `configuration → config → Extensions → Typelist` and opens it in the Typelist editor.
4. In `ContactRel.ttx`, right-click an existing typecode and choose `Add new → typecode`.
5. Enter the following values for the new typecode:
  - **code** – `primaryinsured`
  - **name** – Primary Insured
  - **desc** – Primary insured
6. In the Project window, navigate to `configuration → config → Extensions → Typelist` and double-click `ContactBidiRel.ttx` to edit it.
7. In `ContactBidiRel.ttx`, right-click an existing typecode and choose `Add new → typecode`.
8. Enter the following values for the new typecode:
  - **code** – `primaryinsured`
  - **name** – Primary Insured
  - **desc** – Primary insured
9. Right-click the new `primaryinsured` typecode and choose `Duplicate`
10. Enter the following values for the duplicated typecode:
  - **code** – `secondaryinsured`
  - **name** – Secondary Insured
  - **desc** – Secondary insured

**11.** In the `contact-relationship-config.xml` file, add elements to combine the relationship `primaryinsured` with its inverse, `secondaryinsured`, as follows:

- a. In the Project window, navigate to `configuration → config → addressbook` and double-click `contact-relationship-config.xml` to open it in the editor.

- b. Add the following elements before the final `</ContactRelationshipConfigFile>` tag:

```
<ContactRelationshipPair contactRelCode="primaryinsured" >
 <Primary name="PrimaryInsured"
 cardinality="zeroorone"
 contactBidiRelCode="primaryinsured"
 entity="Contact" />
 <Inverse name="SecondaryInsured"
 cardinality="zeroormore"
 contactBidiRelCode="secondaryinsured"
 entity="Contact" />
</ContactRelationshipPair>
```

**12.** Ensure that your synchronization configuration also has the primary relationship defined as a synchronization attribute. Add the relationship to Contact as follows:

**13.** Press **CTRL+N** and enter `RelationshipSyncConfig`, and then double-click the class name in the search results.

**14.** Add the following chained method call to the `init` method before the final `.create` method call:

```
.includeRelationship(Contact, TC_PRIMARYINSURED, true)
```

For information on using `RelationshipSyncConfig`, see “Setting Relationships to Be Included or Excluded” on page 204 in the *Contact Management Guide*.

**15.** Stop ClaimCenter, regenerate the data dictionary, and then restart ClaimCenter, as described in the following steps. You need to do all these steps because the relationship additions cause data model changes to `Contact`.

- a. Open a command prompt in the `ClaimCenter/bin` directory and then enter the following command:

```
gwcc dev-stop
```

- b. Regenerate the data dictionary to ensure that your changes are correct:

```
gwcc regen-dictionary
```

- c. Start the ClaimCenter application:

```
gwcc dev-start
```

**16.** To test that the changes work:

- a. Log in as a user that can edit claims. For example, if using the sample data, use `aapplegate/gw`.

- b. Open an existing claim and open the **Parties Involved** → **Contacts** screen.

- c. If necessary, add contacts to this screen so that there are at least two.

- d. Click the name of one of the contacts to open the **Basics** card below, then click the **Related Contacts** card.

- e. Click **Edit**, then **Add**, and then click the new **Name** field to add one of the contacts on the list.

- f. Under **Relation to Contact**, click the field and choose **Primary Insured** from the drop-down list.

- g. Click **Update** to save your changes.

- h. In the list of contacts above the **Related Contacts** card, click the contact you just added as a related contact, then click the **Related Contacts** card for that contact.

That tab shows the first contact you picked and the inverse relationship **Secondary Insured**.

#### To add the same new relationship to Contact Manager

1. Start Contact Manager Studio.

At a command prompt, navigate to the Contact Manager installation `bin` directly and enter the following command:

```
gwab studio
```

2. In the Project window, navigate to **configuration** → **config** → **Extensions** → **TypeList** and double-click **ContactRel.ttx** to edit it.
3. In **ContactRel.ttx**, right-click an existing typecode and choose **Add new** → **typecode**.
4. Enter the following values for the new typecode:
  - **code** – **primaryinsured**
  - **name** – Primary Insured
  - **desc** – Primary insured
5. In the Project window, in the same **TypeList** folder, double-click **ContactBidiRel.ttx** to edit it.
6. In **ContactBidiRel.ttx**, right-click an existing typecode and choose **Add new** → **typecode**.
7. Enter the following values for the new typecode:
  - **code** – **primaryinsured**
  - **name** – Primary Insured
  - **desc** – Primary insured
8. Right-click the new **primaryinsured** typecode and choose **Duplicate**
9. Enter the following values for the duplicated typecode:
  - **code** – **secondaryinsured**
  - **name** – Secondary Insured
  - **desc** – Secondary insured
10. In the **abcontact-relationship-config.xml** file, add elements to combine the relationship **primaryinsured** with its inverse, **secondaryinsured**, as follows:
  - a. In the Project window, navigate to **configuration** → **config** → **addressbook** and double-click **abcontact-relationship-config.xml** to open it in the editor.
  - b. Add the following elements before the final **</ContactRelationshipConfigFile>** tag:

```
<ContactRelationshipPair contactRelCode="primaryinsured" >
 <Primary name="PrimaryInsured"
 cardinality="zeroorone"
 contactBidiRelCode="primaryinsured"
 entity="ABContact" />
 <Inverse name="SecondaryInsured"
 cardinality="zeroormore"
 contactBidiRelCode="secondaryinsured"
 entity="ABContact" />
</ContactRelationshipPair>
```
11. Stop Contact Manager, regenerate the data dictionary, and restart Contact Manager, as described in the steps that follow.

You need to do all this because the relationship additions cause data model changes to **ABContact**.

  - a. If necessary, stop the Contact Manager application:  
Open a command prompt in the **ContactManager/bin** directory and then enter the following command:  
`gwab dev-stop`
  - b. Regenerate the data dictionary to ensure that your changes are correct:  
`gwab regen-dictionary`
  - c. Start the Contact Manager application:  
`gwab dev-start`

#### To test that the changes work

1. Log into ClaimCenter as a user such as the sample user **ssmith/gw**.

2. Open an existing claim and open the **Parties Involved** → **Contacts** screen.
3. Click **Add existing contact** and search for a contact, such as the company **Whole Foods** from the imported sample data.
4. Select the contact and add a role so you can add it to the list of contacts for the claim.
5. Click **Update** to add the contact to the claim.
6. Select the newly added contact and then click the **Related Contacts** card.
7. Click **Edit** and then click **Add** to add a new related contact.
8. Click the drop-down button in the **Name** field and choose **Search** from the list, and then search for an existing contact. For example, search for the Person with **First Name** of **Eric** and **Last Name** of **Andy** from the imported sample data.
9. Click **Select** next to the contact to want to add.
10. On the **Related Contacts** card in the **Relationship to Contact** field, click the field and choose **Primary Insured** from the drop-down list.
11. Click **Update** to save your changes.
12. In the **Related Contacts** card for your contact, click the **Name Eric Andy**.
13. Click the **Related Contacts** tab. That tab shows the first contact you picked, such as **Whole Foods**, and the inverse relationship **Secondary Insured**.

**See also**

- “Adding Contact Roles” on page 587
- “How Configuring Roles Impacts Entity Data and Types” on page 591
- “Adding a New Contact Role: an Example” on page 594
- “Synchronizing ClaimCenter and ContactManager Contacts” on page 200 in the *Contact Management Guide*

# Configuring Multicurrency

This topic describes multicurrency configuration in ClaimCenter.

This topic includes:

- “Multicurrency in Financial Calculations” on page 601
- “Multicurrency Data Model” on page 602
- “Foreign Exchange Adjustments” on page 604
- “Foreign Exchange Transactions and Calculated Values” on page 605
- “Foreign Exchange Adjustments on Claims and Payments” on page 605

**See also**

- “Configuring Currencies” on page 85 in the *Globalization Guide*
- “Exchange Rate Integration” on page 409 in the *Integration Guide*

## Multicurrency in Financial Calculations

ClaimCenter supports multicurrency transactions. You can carry out transactions (create reserves and recovery reserves, make payments, and collect recoveries), as well as write checks, in more than one currency in a single claim. ClaimCenter supports a single default currency and multiple secondary transaction currencies:

Default currency	ClaimCenter defines its default currency with the configuration parameter <code>DefaultApplicationCurrency</code> (in <code>config.xml</code> ) on server startup. Currently, the terms <i>claim currency</i> and <i>reporting currency</i> also refer to this default currency. <b>Note:</b> The <code>ExchangeRate</code> entity contains a <code>BaseCurrency</code> typekey. Do not confuse this with the default currency.
Claim currency	The claim's currency, inherited from the associated policy, which is defined in the <code>Currency</code> typelist in the <code>Claim</code> entity.

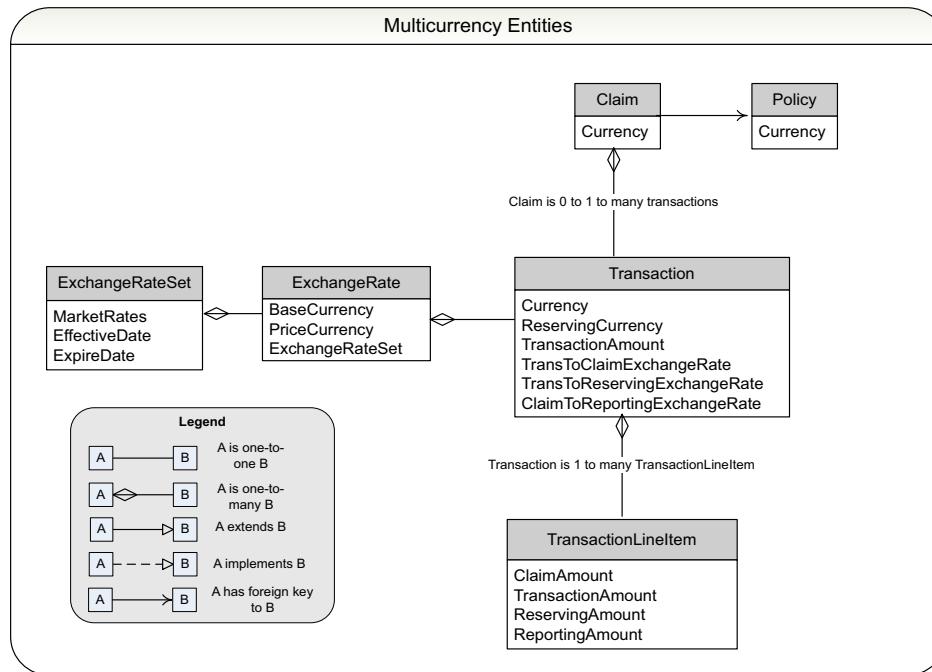
Reserving currency	Currency of a reserve line, defined by the ReservingCurrency typelist in the ReserveLine entity.
Transaction currencies	ClaimCenter maintains a list of all of the available currencies in the Currency typelist.

#### See also

- For information on how ClaimCenter handles multiple currencies, see “Multicurrency Overview” on page 335 in the *Application Guide*.
- For information on using methods on `IClaimFinancialsAPI` that are specific to foreign exchange adjustments, see “Claim Financials Web Services (ClaimFinancialsAPI)” on page 369 in the *Integration Guide*.

## Multicurrency Data Model

The following section summarize the main entities related to multicurrency.



- The currency of the claim always equals the currency of the policy.
- There is a foreign key from **Transaction** to **Claim**.
- TransactionLineItem** stores amounts in all four currencies that affect a transaction: transaction, claim, reserving, and reporting. The **TransactionLineItem** table itself only stores the amounts. The currencies are stored in the **Transaction** and **Claim** entities respectively and statically in the system, as configured in the `config.xml` file. The `ReportingAmount` is from the application’s default currency. ClaimCenter stores this currency as a configuration parameter in the `config.xml` file.

- The transaction-to-claim, transaction-to-reserving, and claim-to-reporting exchange rates are stored in the corresponding fields in Transaction.

Entity	Description
Check	<p>Contains an array of Payments. Payment is a subtype of Transaction, so every Payment stores a currency and two exchange rates. All the payments on a check must have the same currency and exchange rates. For this reason, the currency for a payment can also be called the <i>check currency</i>. The Check entity does not store a currency or any exchange rates, but Check does have virtual properties to get the currency and exchange rates for its payments.</p>
CheckPortion	<p>Determines the amount for which a secondary check will be written. For example, a secondary check could be a check on a multi-payee check that is not the primary check. CheckPortion can indicate that a secondary check receives either a percentage of the sum of the payments or a fixed amount.</p> <p>Checks with a CheckPortion do not have any payments, but just receive a percentage or fixed amount of the Payments in the primary Check of the CheckGroup. CheckPortion cannot have its percentage and fixed amount fields both populated.</p> <p>Important fields include:</p> <ul style="list-style-type: none"> <li>FixedTransactionAmount – If set, the amount in the transaction currency that is allocated to this secondary check.</li> <li>FixedClaimAmount – If set, the amount in the claim currency that is allocated to this secondary check.</li> <li>FixedReportingAmount – If set, the amount in the reporting currency that is allocated to this secondary check.</li> <li>Percentage – If set, the fraction of the amounts of the payments that are allocated to this secondary check. Setting this field clears the fixed amount fields.</li> </ul>
Deduction	<p>Represents a deduction from a check, usually for tax purposes.</p> <p>Important fields include:</p> <ul style="list-style-type: none"> <li>ClaimAmount – Amount of the deduction in the claim currency.</li> <li>TransactionAmount – Amount of the deduction in the transaction currency.</li> <li>ReportingAmount – Amount of the deduction in the reporting currency.</li> </ul>
Transaction	<p>Represents a financial operation. It has the following subtypes: Payment, Recovery, RecoveryReserve, and Reserve.</p> <p>Important fields include:</p> <ul style="list-style-type: none"> <li>Currency – Transaction currency.</li> <li>TransToReservingExchangeRate – Exchange rate between transaction and reserving currencies that is in effect for this Transaction.</li> <li>TransToClaimExchangeRate – Exchange rate between transaction and claim currencies that is in effect for this Transaction.</li> <li>ClaimToReportingExchangeRate – Exchange rate between claim and reporting currencies that is in effect for this Transaction. This field is not shown in the user interface in the base configuration.</li> </ul>
TransactionLineItem	<p>TransactionLineItems provide a means to split the amount of a transaction into multiple categories such as Doctor, Hospital, Legal, and so forth. Every transaction has one or more TransactionLineItems, and the amount of the transaction is the sum of all its line items' amounts.</p> <p>Important fields include:</p> <ul style="list-style-type: none"> <li>Deductible – Deductible represented by this TransactionLineItem, if any. Used when deductible handling is enabled.</li> <li>TransactionAmount, ClaimAmount, ReservingAmount, ReportingAmount – Store the amount in the four currencies. TransactionAmount is also accessible through the Amount virtual property for single currency implementations.</li> </ul>

Entity	Description
ExchangeRate	<p>Represents an exchange rate between a pair of currencies. This rate can be a market rate, in which case it will exist in an ExchangeRateSet with rates between every currency pair. It can also be a manually-entered custom rate, in which case it typically contains the amount entered by the user and resides alone in an ExchangeRateSet.</p> <p>Important fields include:</p> <ul style="list-style-type: none"> <li>• BaseCurrency – The currency from which this exchange rate converts, as in GB pounds.</li> <li>• PriceCurrency – The currency to which this exchange rate converts, as in Euros.</li> <li>• Rate – The exchange rate between the two currencies, such as 1.301.</li> </ul>
ExchangeRateSet	<p>Represents a set of exchange rates, along with supplemental information about those rates, including the dates the set goes into effect and expires.</p> <p>The MarketRates field, when true, indicates that the exchange rates are market rates. When this field is false, the set contains only one user-entered custom rate.</p> <p>Important fields include:</p> <ul style="list-style-type: none"> <li>• EffectiveDate – Sets the date and time at which the rate set starts to be in effect.</li> <li>• ExpireDate – Sets the date and time at which the rate set becomes no longer in effect.</li> <li>• MarketRates – If set to true, the rate set is included in the search for the latest market rates.</li> </ul>

## Foreign Exchange Adjustments

For foreign-currency payments, it is possible that the exchange rate can change during a transaction. For example, the exchange rate between the payment's currency and the claim's base currency can change. This can happen, perhaps, in between the time that the payment was created and the time its associated check finally clears the bank. For this reason, ClaimCenter provides the ability to update the converted amount of a check and its payments once the final converted amount is known.

ClaimCenter supports this update process by having the user specify the final converted amount of the check or its payments, as opposed to the final exchange rate. ClaimCenter then uses this final converted amount to calculate a *foreign exchange adjustment* for the payments. It then updates the underlying accounts appropriately. ClaimCenter makes the adjustment for each payment at the level of the `TransactionLineItem`, making each adjustment proportional to the line items' amounts.

For example, suppose that you have a check with a single payment that has three line-items with the following transaction and claim amounts:

Line-item	Transaction Amount	Claim Amount
1	60	50
2	35	30
3	22	20
<b>Total Amount</b>	<b>117</b>	<b>100</b>

At some future time, the payment's check clears, and the converted claim amount is actually \$110, rather than \$100. ClaimCenter applies the foreign exchange adjustment proportionally across all the line-items, resulting in the following:

Line-item	Transaction amount	Claim amount	Foreign Exchange Adjustment	Cleared Claim Amount
1	60	50	5	55
2	35	30	3	33

Line-item	Transaction amount	Claim amount	Foreign Exchange Adjustment	Cleared Claim Amount
3	22	20	2	22
<b>Total Amount</b>	<b>117</b>	<b>100</b>	<b>10</b>	<b>110</b>

ClaimCenter provides methods for foreign exchange adjustments to be made on either a payment-by-payment basis, or for an entire check at once. In the latter case, the adjustment is split proportionally between all the active payments on the check (excluding any recoded, offsetting or canceled payments). Then, for each such payment, its adjustment is split amongst its line-items as discussed in the previous example.

Applying a foreign exchange adjustment to an existing payment or check is equivalent to setting a custom exchange rate on the payment or payments.

## Foreign Exchange Transactions and Calculated Values

Guidewire ClaimCenter treats foreign exchange adjustments as non-eroding. This means the following:

- Foreign exchange adjustments *do not* decrease calculated values such as Open, Remaining and Available Reserves.
- Foreign exchange adjustments *do* increase all total incurred calculations (Net and Gross), as well as overall Total Paid and Total Paid Non-Eroding.

In the example listed in “Foreign Exchange Adjustments” on page 604, assume that the only other transaction on the payment's reserve line is a reserve transaction for \$150. (This transaction is in the base currency.) As the payment is first escalated, you see the following calculated values:

Open, Remaining, and Available Reserves	\$50.00
Total Incurred (gross and net)	\$150.00
Total Paid	\$100.00
Total Paid Non-eroding	\$0.00
Total Paid Eroding	\$100.00

After ClaimCenter applies the foreign exchange adjustment, the calculated values become:

Open, Remaining and Available Reserves	\$50.00 (unchanged)
Total Incurred (gross and net)	\$160.00 (+\$10.00)
Total Paid	\$110.00 (+\$10.00)
Total Paid Non-eroding	\$10.00 (+\$10.00)
Total Paid Eroding	\$100.00 (unchanged)

To work with foreign exchange calculations, Guidewire provides three expressions accessible from `gw.api.financials.FinancialsCalculationUtil`:

```
getForeignExchangeAdjustmentsExpression
getErodingPaymentsForeignExchangeAdjustmentsExpression
getNonErodingPaymentsForeignExchangeAdjustmentsExpression
```

## Foreign Exchange Adjustments on Claims and Payments

**Note:** Foreign exchange adjustments only affect total incurred and total paid calculations. They do not further erode reserves.

Guidewire ClaimCenter provides the following methods to apply a foreign exchange adjustment to a check or a payment within the context of the ClaimCenter business rules:

- `Check.applyForeignExchangeAdjustment`
- `Payment.applyForeignExchangeAdjustment`

#### **Check.applyForeignExchangeAdjustment**

```
Check.applyForeignExchangeAdjustment(newClaimAmount : BigDecimal) : void
```

This method applies a foreign exchange adjustment to this Check's claim currency amount. The parameter `newClaimAmount` specifies the new amount for this check in the claim currency. It cannot be `null`.

The amount of the adjustment is the percentage difference between the new passed-in amount and the current amount for claim currency. Suppose, for example, that the original check has three payments of \$50, \$20 and \$10 for a total claim amount of \$80, and the new claim amount is \$100. Then, every payment will get a 25% offset, making them \$62.50, \$25 and \$12.50, for a total of \$100.

Only use this method on a check that is in a committed but uncanceled state, meaning that the check status must be one of the following:

- Requesting
- Requested
- Issued
- Cleared

#### **Payment.applyForeignExchangeAdjustment**

```
Payment.applyForeignExchangeAdjustment(newClaimAmount : BigDecimal) : void
```

This method applies a foreign exchange adjustment to this Payment's claim currency amount. The parameter `newClaimAmount` specifies the new amount for this payment in the claim currency. It cannot be `null`.

The amount of the adjustment is the percentage difference between the new passed-in amount and the current amount for claim currency. Suppose, for example, that the original payment has three line items of \$50, \$20 and \$10 for a total claim amount of \$80, and the new claim amount is \$100. Then, every line-item will get a 25% offset, making them \$62.50, \$25 and \$12.50, for a total of \$100.

Only use this method on a payment that is an offsetting payment. The payment must also be in a committed but uncanceled state, meaning that the payment status must be one of the following:

- Submitting
- Submitted

### Applying Foreign Exchange Adjustments Multiple Times

You can apply a foreign exchange adjustment to a payment or check multiple times. However, each application of the adjustment negates and replaces the prior one. Using the previous example in “Foreign Exchange Adjustments” on page 604 (three payments of \$50, \$30, and \$20), assume that you make a second call to `applyForeignExchangeAdjustment` on the payment. This time, you pass in a value of \$120.00. You would then end up with the following three line-items:

- \$60.00 (\$10.00 adjustment)
- \$36.00 (\$6.00 adjustment)
- \$24.00 (\$4.00 adjustment)

The ClaimCenter database maintains a history of both of these adjustments in order to maintain an audit trail. However, it only applies the last adjustment to calculated values.

## TransactionLineItem Fields

ClaimCenter provides the following fields on the `TransactionLineItem` entity for use with foreign exchange adjustments:

- `ClaimForExAmount`
- `ReportingForExAmount`

It also provides the following fields on the `Transaction` entity:

- `ClaimForExAdjustmentAmount`
- `ReportingForExAdjustmentAmount`



---

part IX

# Guidewire ClaimCenter Financials



# Configuring ClaimCenter Financials

This topic explains the data model and financial values managed by the ClaimCenter Financials. It explains how to configure both the financials behavior and the ClaimCenter interface to better match your business practices.

This topic includes:

- “Overview of the Financials Data Model” on page 612
- “Financial-related Typelists” on page 613
- “Financial Transaction Statuses” on page 614
- “Financial Configuration Parameters” on page 615
- “Batch Processes Related to Checks and Payments” on page 616

## Overview of the Financials Data Model

The following table contains the key financials entities in the data model that you see in the ClaimCenter base configuration. Refer to the *ClaimCenter Data Dictionary* to see other financial-related entities.

Entity or field	Description
Check	An entity that groups one or more payments made at the same time to a single payee or group of joint payees. ClaimCenter sends it to an external system to be printed, unless it is a manual check not created by the application.
CheckGroup	An entity that groups together a multipayee check, with a primary check and one or more secondary checks.
CheckSet	The entity that collects all Checks resulting from a single usage of the New Check wizard. It includes all issuances of a recurring Check and checks of a multipayee Check. It is a subtype of TransactionSet. All Checks belong to a CheckSet.
CostCategory	A Transaction field that categorizes a transaction. In the base configuration, the CostCategory typelist includes values that you can use as filters to support the various Lines of Business (LOBs)
CostType	A Transaction field that categorizes a transaction. In the base configuration, the CostType typelist includes the following typecodes: <ul style="list-style-type: none"> <li>• aoexpense – Adjusting and other expense</li> <li>• claimcost – Actual loss payments to claimants or repairers</li> <li>• dccexpense – Defense and cost containment legal expense</li> <li>• unspecified – Unspecified cost type</li> </ul>
Deductible	The entity that tracks the amount, the coverage, and the status of the deductible, such as whether it has been paid or waived. One of the main fields on the Deductible entity is TransactionLineItem, which is a foreign key to TransactionLineItem.
Line Category	A field in a TransactionLineItem that categorizes the amount of that line item.
Payment	A subtype of Transaction representing money paid out. A payment can be eroding or non-eroding, depending on whether it draws down the reserves of its ReserveLine.
Recovery	An entity that records money that reduces a claim's liability, received from such sources as subrogation, salvage, other insurance, co-payments or deductibles. A Recovery object is a subtype of Transaction.
RecoveryReserve	An entity that records the amount of future expected recoveries. It is a subtype of Transaction.
Reserve	An entity that records a potential liability. It is a subtype of Transaction. A Reserve designates money to be set aside for payments. Typically, a reserve is set soon after a claim is made.
ReserveLine	An entity with a unique combination of Claim, Exposure, CostType, and CostCategory fields. Only Exposure can be null. Reserves or recovery reserves are created, or payments are made, or recoveries are applied against one ReserveLine.
Transaction	An entity that represents a financial transaction for a particular claim or exposure. It also contains a non-empty array of TransactionLineItem entities. <p>Transaction is an abstract supertype. The ClaimCenter interface uses its subtypes:</p> <ul style="list-style-type: none"> <li>• Reserve</li> <li>• Payment</li> <li>• RecoveryReserve</li> <li>• Recovery</li> </ul> <p>Every transaction is made against a single ReserveLine object.</p>
TransactionLineItem	An entity in every transaction that contains the amount of the transaction. Payment and Recovery transactions can have more than one Transaction Line Item. Use the LineCategory and Comments fields to describe a given Transaction Line Item's contribution to the total transaction amount.
TransactionOnset	This join entity contains a foreign key to the Transaction entity and represents the relationship between a transaction and its onset. It links a Transferred or Recoded transaction (Payment or Recovery) to its new onset transaction.

Entity or field	Description
TransactionOffset	This join entity contains a foreign key to the Transaction entity and represents the relationship between a transaction and its Offset. It links a Voided, Stopped, Recoded, or Transferred transaction (Payment or Recovery) to its new onset transaction.
TransactionSet	A collection of all transactions made at the same time and approved together. This collection can be, for example, a check and all the payments it makes. TransactionSet is an abstract supertype. The ClaimCenter interface uses the following sub-types of TransactionSet: <ul style="list-style-type: none"> <li>• ReserveSet</li> <li>• CheckSet</li> <li>• RecoveryReserveSet</li> <li>• RecoverySet</li> </ul> CheckSet is a subtype of TransactionSet. A check is not a Transaction. The checks in the set, while created at the same time, can be issued at different times and to different payees. You can also associate documents with a TransactionSet. All transactions (and checks) in a Transaction Set must be: <ul style="list-style-type: none"> <li>• Approved together</li> <li>• Rejected together</li> <li>• In Pending Approval status together</li> </ul>

Transaction entities are the key to understanding ClaimCenter financials. A transaction is linked either to a claim or to an exposure (and every exposure is linked to a claim). A Transaction can have a CostType attribute whose value is non-nullable. You can use the default cost types or create others that reflect your business process. In the base configuration, ClaimCenter provides the following cost types:

aoexpense	Overhead costs such as adjusting and other expenses.
claimcost	Indemnity losses—actual loss payments to claimants or repairers.
dcexpense	Legal costs for defense and cost containment.
unspecified	Unspecified cost type

ClaimCenter applies a transaction against a specific ReserveLine. A ReserveLine is a unique combination of Claim, Exposure, CostType, and CostCategory.

A Transaction must have at least one—and can have more—TransactionLineItem objects. The TransactionLineItem object contains an actual value amount. The value of an individual Transaction is the sum of its TransactionLineItem amounts. Each TransactionLineItem has an associated LineCategory that further classifies its cost type, for example doctor's care, physical therapy, and so forth. You can configure the Line Categories to reflect your business practices.

#### See also

- “Financials Data Model” on page 330 in the *Application Guide*

## Financial-related Typelists

The following typelists contain values that you can modify to manipulate the financial configuration in your ClaimCenter installation:

Typelist	Description
BankAccount	Lists bank accounts available from the ClaimCenter interface.
BiValidationAlertType	Lists the possible alert types returned from the bulk invoice validation plugin.

TypeList	Description
CheckBatching	ClaimCenter populates the <b>Check Batching</b> drop-down list on the check wizard from this list.  <b>Note:</b> The <code>bulkcheck</code> typecode on this typelist has nothing to do with Bulk Invoices.
CheckHandlingInstructions	Specifies handling instructions for a check. ClaimCenter populates the <b>Check Instructions</b> drop-down list on the check wizard from this list.
CostCategory	Categories for different costs associated with financial transactions.
CostType	Defines different costs associated with your business. ClaimCenter uses the values in this typelist as a key filter for the <code>CostCategory</code> typelist.
CoverageType	Types of coverage available on a claim. ClaimCenter uses the values in this typelist as a key filter for the <code>CostCategory</code> typelist.
DeductionType	Types of deductions that can appear on a check. The typical use for this typelist is to categorize both a <code>Deduction</code> entity and a secondary check on a multipayee check.
FinancialSearchField	Search fields used while searching for checks or recoveries.
LineCategory	Populates the <b>Category</b> drop-down on the <b>Payment Information</b> page of the check wizard. Use to categorize the <b>Amount</b> of a Transaction Line Item.
PaymentMethod	Requested payment method for all payments in a check. For example, use to set whether to send the check as a paper check or EFT.

## Financial Transaction Statuses

You can think of a financial transaction as having a life cycle. During its life cycle, as a transaction transitions from one state to another, the `TransactionStatus` attribute on the transaction changes. The property on `Transaction` that stores the status is called `Status`. This is a typekey in the `TransactionStatus` typelist. Review the `TrasactionStatus` typelist in the *Data Dictionary* for a full list of the possible transaction statuses.

You can access the `Transaction.Status` property from both the ClaimCenter interface and from within Gosu rules. Its primary use is to indicate the current business state of a transaction, as well as act on the business state of the transaction.

The following are all valid values for the `Transaction.Status` property.

- Awaiting submission
- Cleared
- Denied (checks and payments only)
- Issued
- Notifying (manual checks only)
- Pending approval
- Pending recode
- Pending stop
- Pending transfer
- Pending void
- Recoded (payments and recoveries only)
- Rejected
- Requested (checks only)
- Requesting (checks only)
- Stopped (payments only)
- Submitted
- Submitting

- Transferred (payments, checks, and recoveries only)
- Voided (payments, checks, and recoveries only)

### Submitted Transactions

Submitted transactions are a subset of Approved transactions. A transaction is considered to be **Submitted** if its Status is any of the following:

- |                    |              |               |
|--------------------|--------------|---------------|
| • Pending          | • Recoded    | • Transferred |
| • Pending Stop     | • Stopped    | • Voided      |
| • Pending Transfer | • Submitted  |               |
| • Pending Void     | • Submitting |               |

### Awaiting Submission Transactions

A transaction is considered to be **Awaiting Submission** if its Status is **AwaitingSubmission**. Only payments and reserves can have this status:

- *AwaitingSubmission payments* are those payments whose containing Check has not yet been escalated.
- *AwaitingSubmission reserves* are those reserves that are tied to an **AwaitingSubmission** payment in order to act as an offsetting reserve.

Payments with a Status of **AwaitingSubmission** can also be future-dated. *FutureDated payments* are those approved payments whose containing Check has a scheduled send date after the current day.

### Pending Approval Transactions

A transaction is considered to be **Pending Approval** if its Status is **PendingApproval**.

### Approved Transactions

A transaction is considered to be **Approved** if its Status is any of the following:

- |                      |                |               |
|----------------------|----------------|---------------|
| • AwaitingSubmission | • Pending void | • Submitting  |
| • Pending recode     | • Recoded      | • Transferred |
| • Pending stop       | • Stopped      | • Voided      |
| • Pending transfer   | • Submitted    |               |

### See also

- “Lifecycles of Transactions” on page 319 in the *Application Guide*

## Financial Configuration Parameters

Guidewire provides a number of configuration parameters related to ClaimCenter financials. These include the following:

- |                                     |                                         |
|-------------------------------------|-----------------------------------------|
| • AllowMultipleLineItems            | • ExchangeRatesCacheRefreshIntervalSecs |
| • AllowMultiplePayments             | • Financials                            |
| • AllowNegativeManualChecks         | • MulticurrencyDisplayMode              |
| • AllowNoPriorPaymentSupplement     | • PaymentLogThreshold                   |
| • AllowPaymentsExceedReservesLimits | • PaymentRoundingMode                   |
| • CheckAuthorityLimits              | • ReserveRoundingMode                   |

- `CloseClaimAfterFinalPayment`
- `CloseExposureAfterFinalPayment`
- `DefaultApplicationCurrency`
- `DefaultRoundingMode`
- `SetReservesByTotalIncurred`
- `UseDeductibleHandling`
- `UseRecoveryReserves`

See “Financial Parameters” on page 58 for a discussion of the parameters related to financials.

## Batch Processes Related to Checks and Payments

There are three batch process that ClaimCenter runs automatically to process financial transactions. This topic describes how those batch processes impact checks and payments.

For more information, see the following:

<i>Checks and payments</i>	“Checks” on page 301 in the <i>Application Guide</i> and “Payments” on page 296 in the <i>Application Guide</i>
<i>Financial batch processes</i>	“List of Batch Processes and Distributable Work Queues” on page 129 in the <i>System Administration Guide</i>
<i>Scheduling batch processes</i>	“Scheduling Batch Processes and Work Queues” on page 143 in the <i>System Administration Guide</i>
<i>Integrating with a check-writing system</i>	“Financials Integration” on page 363 in the <i>Integration Guide</i>

### The Financials Escalation Process

ClaimCenter periodically runs a `financialsesc` batch process that looks for checks that are ready for submission. A check is automatically eligible for submission:

- If it has a `TransactionStatus` of `awaitingsubmission`
- If it has a scheduled send date that is either *today* or earlier
- If it is not part of a bulk invoice

**Note:** If a check is marked for `PendEscalationForBulk`, then it will *not* be submitted.

As ClaimCenter escalates a check through the `financialsesc` process, it does the following:

- ClaimCenter updates all the associated T-accounts, unless the `taccountesc` process has already run and performed this task.
- ClaimCenter creates any necessary offsetting reserves. This and any other associated reserve changes are given `submitting` status. For example, if an eroding payment exceeds its available reserves, it requires an offset to keep its available reserves from becoming negative (unless `taccountesc` has already run and done this).
- ClaimCenter changes the `TransactionStatus` for the check to `requesting`. At this point, it is possible to send a message to issue the check to a check writing system.
- ClaimCenter changes the `TransactionStatus` on all the associated payments to `submitting`.
- If a payment being escalated is final:
  - *And the payment has an exposure*, then ClaimCenter closes its exposure, provided it has no payments scheduled for sending after today.
  - *And the payment is claim-level*, then ClaimCenter closes its claim, provided the claim has no open exposures and no payments scheduled for sending after today.

If either of these close actions result in a validation error or warning, ClaimCenter creates a reminder activity showing the error or errors. It then tries to assign the activity to the user that created the payment. If that fails,

it uses auto-assignment to assign the activity. The due date for the activity is today, its priority is normal, and no escalation date is set.

- ClaimCenter runs the Transaction Post-Setup rules.
- If the check is a recurring check and it is the second-to-last check to be submitted in the recurrence, then ClaimCenter creates a reminder activity. The activity simply indicates that the recurrence is ending soon.

By default the `financialsesc` process runs twice a day. It runs the first time at 6:05 a.m. and the second time at 6:05 p.m. To change this value, edit the `scheduler-config.xml` file. Alternatively, you can also start the process from the command line to force it to run immediately.

```
maintenance_tools -startprocess financialsesc -password <password> -user <username>
```

You can also create a Gosu rule to escalate a check by using the `Check.requestCheck` method.

**Note:** If entering the date for escalation, enter a day only but not a time. If a time is present, the batch process delays escalation until the first time it runs on the next day.

## The TAccounts Escalation Process

The `TAccountEsc` batch process updates T-accounts and summary financial values to reflect the fact that a check is going to be issued on that date, without the check being issued. The process moves a Check's payments from a Future Dated state to an AwaitingSubmission today state. (The Status remains AwaitingSubmission.)

Financials calculations such as Total Payments and Open Reserves include payments waiting to be sent today. Thus, the `TAccountEsc` process updates those calculations shown in the **Financials Summary** screen to reflect payments that will now be sent today. However, since the Checks are still in Awaiting Submission, it is still possible to edit them until they are escalated by the `financialsesc` batch process.

When a payment contributes to Open Reserves by eroding reserves, it can make Open Reserve negative. In this case, ClaimCenter automatically creates offsetting reserves during this batch process to keep OpenReserves at zero.

The `TAccountEsc` batch process does the following:

- It correctly updates the relevant T-accounts to reflect the fact that the check reached its send date.
- It creates any necessary zeroing-offsets for the payments for each check (if the payment exceeds reserves). This and any other associated reserve changes are given `awaiting submission` status, which means that they can still be retired if their associated payments are retired or changed. For example, if an eroding payment exceeds its available reserves, it requires an offset to keep its available reserves from becoming negative.

By default, `taccountesc` runs at 12:01 a.m. every day. You can reschedule the `taccountesc` batch process by editing the `scheduler-config.xml` file. You can also start the process from the command line to force it to run immediately.

```
maintenance_tools -startprocess taccountesc -password <password> -username <username>
```

---

**IMPORTANT** The `taccountesc` batch process works with the T-account balances and the calculated financials values that depend on these balances. It ensures that these balances are correct during the interval between midnight and the first scheduled execution of the `financialsesc` batch process for the day. Guidewire recommends that you schedule the `taccountesc` batch process to run as close to just-past midnight as possible and *before* the `financialsesc` batch process.

---

## The Bulk Invoice Escalation Process

As its name implies, the `bulkinvoiceesc` process escalates the status of bulk invoices in your installation. If this process runs, it does the following:

- Queries for all bulk invoices with a status of `awaitingsubmission` and a scheduled send date of the current day or earlier.
- Sets the status of each returned bulk invoice to `requesting`.

- Locates all of the invoice items for which an associated check exists. If the check property `pendescalationforbulk==true` and the check `status==awaitingsubmission`, then the process escalates the check also. This has the effect of also moving the invoice item to the `submitting` status.

By default, the `bulkinvoiceesc` process runs every day at 6:35 a.m. and 6:35 p.m. You can reschedule the `bulkinvoiceesc` batch process by editing the `scheduler-config.xml` file. Alternatively, you can also start the process from the command line to force it to run immediately.

```
maintenance_tools -startprocess bulkinvoiceesc -password <password> -username <username>
```

**Note:** Do not schedule the `bulkinvoiceesc` process to run at the same time as the `financialsesc` process. Running these two at the same time makes both processes take much longer than necessary to complete.

### The PendEscalationForBulk Property

Checks associated with a bulk invoice are normally escalated whenever the Bulk Invoice is escalated. If `PendEscalationForBulk` on a check is set to `true`, then batch process `bulkinvoiceesc` escalates the check. You can set `Check.PendEscalationForBulk` to `false`, for example, if your downstream system automatically batches together checks to the same payee, and you want `BIIItem` checks escalated individually.

This property has the following meanings:

- `PendEscalationForBulk==true`. Guidewire sets this property to `true` by default, upon the initial creation of a Check for a `BIIItem`. A value of `true` means that ClaimCenter escalates the Check at the same time as the `bulkinvoiceesc` batch process escalates the `BulkInvoice`.
- `PendEscalationForBulk==false`. It is possible to change the value of this property to `false` through Gosu rules. A value of `false` indicates that ClaimCenter escalates the Check as part of the standard `financialsesc` batch process.

Another property, `Check.Bulked`, indicates whether ClaimCenter associates a Check with a `BulkInvoiceItem`.

The escalation process only updates data inside Guidewire ClaimCenter. Communicating the escalation to an external system is done by actions taken in Event Messaging rules and your messaging plugin implementations.

#### See also

- “Check Integration” on page 372 in the *Integration Guide*.
- “Bulk Invoice Integration” on page 391 in the *Integration Guide*.

### Scheduling the Financials Batch Processes

In the default application, the `financialsesc` batch process runs twice a day, at 6:05 a.m. and 6:05 p.m. If you do not run the `taccountesc` batch process earlier in the day, then the ClaimCenter **Financials Summary** screen can potentially display out-of-date values. The values are out-of-date from midnight until the time that the `financialsesc` batch process does run.

Depending on your implementation, you can schedule these two batch processes differently. For example:

- Schedule one of these two processes to run before the calculated values need to be up-to-date, for example, before people arrive for work.
- Run `taccountesc` as soon after midnight as possible and schedule `financialsesc` at your midday time to keep checks editable until sometime during the last available working day.
- Schedule `financialsesc` just after midnight if you do not care about not being able to edit future-dated checks that have reached their send date before `financialsesc` runs. In this case, you need not run `taccountesc`.

# ClaimCenter Financial Calculations

This topic describes the financial calculation APIs that Guidewire provides in Guidewire ClaimCenter.

This topic includes:

- “Financial Calculation APIs” on page 619
- “Understanding ClaimCenter Financial Calculations” on page 620
- “Using the Predefined Financial Calculations” on page 621
- “Predefined Financial Calculations” on page 623
- “Predefined Reinsurance Calculations” on page 627
- “Retrieving Transaction IDs” on page 628
- “Forming Composite Custom Expressions” on page 629
- “Financial Calculations with a Negative Value” on page 630
- “Eroding and Non-eroding Payments” on page 631
- “Using Floating Financial Calculations” on page 631

**See also**

- “Creating ClaimCenter Financial Transactions” on page 635

## Financial Calculation APIs

Guidewire provides a number of APIs to facilitate various operations with financial calculations. You access these APIs in the following package:

```
gw.api.financials
```

The following list describes these APIs and provides a few sample methods:

<b>gw.api.financials.*</b>	<b>Returns</b>
FinancialsCalculationUtil	<ul style="list-style-type: none"> <li>• FinancialsCalculation objects</li> <li>• FinancialsExpression objects</li> <li>• CurrencyAmount objects for Amounts</li> </ul> <p><b>Sample methods</b></p> <ul style="list-style-type: none"> <li>• <code>getAvailableReserves</code></li> <li>• <code>getFuturePayments</code></li> <li>• <code>getTotalIncurredNet</code></li> <li>• <code>getFinancialsCalculation(FinancialsExpression)</code></li> </ul> <p><b>See also</b></p> <ul style="list-style-type: none"> <li>• “Configuring the Financial Summary Screen” on page 643 for examples of how to use this API</li> </ul>
FinancialsCalculator	<ul style="list-style-type: none"> <li>• FinancialsCalculator objects</li> </ul> <p><b>Sample methods</b></p> <ul style="list-style-type: none"> <li>• <code>getFinancialsCalculation(FinancialsExpression)</code></li> <li>• <code>getFuturePayments</code></li> <li>• <code>getSubmittedOpenReserves</code></li> </ul> <p><b>See also</b></p> <ul style="list-style-type: none"> <li>• “Obtaining Calculated Amounts” on page 621</li> <li>• “Retrieving Transaction IDs” on page 628</li> </ul>
FinancialsExpression	<p>A new composite FinancialsExpression that consists of this FinancialsExpression added to the indicated FinancialsExpression. This composite is a new expression that is the mathematical sum or difference of the values of the two expressions.</p> <p>You can wrap a FinancialsExpression in a FinancialsCalculation or a FinancialsCalculator in order to evaluate its amount.</p> <p><b>Sample methods</b></p> <ul style="list-style-type: none"> <li>• <code>minus(FinancialsExpression)</code></li> <li>• <code>plus(FinancialsExpression)</code></li> </ul> <p><b>See also</b></p> <ul style="list-style-type: none"> <li>• “Forming Composite Custom Expressions” on page 629</li> </ul>

## Understanding ClaimCenter Financial Calculations

ClaimCenter provides the ability to create calculated financial values that can be used anywhere that Gosu is accessible, such as, for example:

- In the ClaimCenter user interface
- In Gosu rules and classes
- In Gosu plugins, and similar places

These calculated values provide various critical views of the financial state of a claim and its exposures.

In the base application configuration, Guidewire provides a set of predefined financial calculations. ClaimCenter uses these financial calculations to compute the values seen on the **Financial Summary** page. For example, you can access the Available Reserves financial expression value by using the following Gosu method:

```
gw.api.financials.FinancialsCalculationUtil.getAvailableReserves
```

The following classes provide methods that you can use to work with financial calculations:

- `gw.api.financials.FinancialCalculationsUtil`
- `gw.api.financials.FinancialsCalculations`

In some cases, they appear to duplicate each other. The difference, however, is in what each method returns. Some return a `FinancialsCalculation` object and others return a `FinancialsCalculator` object.

Guidewire recommends that you primarily use the methods that return a `FinancialsCalculation` object. Use the methods that return a `FinancialsExpression` object only if you want to combine these objects to produce custom expressions from which you can obtain the desired value.

If there are multiple versions of a method, then choose the version that returns the type of object that you need. Generally, you use the predefined financial calculations that Guidewire provides in the base configuration. However, it is possible to define your own financial calculations out of the set of financial calculations that Guidewire provides. For more information, see “[Forming Composite Custom Expressions](#)” on page 629.

## Using the Predefined Financial Calculations

In the base application configuration, Guidewire provides a set of predefined financial calculations. The `FinancialsCalculationUtil` and `FinancialsCalculations` classes provide static methods for financial calculations. Both classes have methods with identical names, but which return different types.

This topic describes the objects used in financial calculations and some of the methods used in retrieving calculated amounts.

### Using a `FinancialsCalculator` Object

A `FinancialsCalculator` is a class in ClaimCenter that provides an alternative way to specify the scope of the calculation total. You use the `gw.api.financials.FinancialsCalculations.*` methods—all of which return a `FinancialsCalculator` object. The `FinancialsCalculator` object contains the following methods that you use to filter the reserve lines that contribute to the calculation total:

- `usesClaimLevelReserves`
- `withAgreement`
- `withClaim`
- `withCostCategory`
- `withCostType`
- `withCoverageType`
- `withExposure`
- `withReserveLine`
- `withRIAgreement`
- `withRIAgreementGroup`
- `withRICoding`

### Obtaining Calculated Amounts

You can use a specialized method on `financials.FinancialsCalculations` to return a `FinancialCalculator` object. For example:

```
gw.api.financials.FinancialsCalculations.getFinancialsCalculation(expression : FinancialExpression)
```

You can also wrap a `FinancialsCalculation` object within a `FinancialsCalculator` object, for example:

```
FinancialsCalculator.onCalculation(someFinancialsCalculation)
```

The end result is a new `FinancialsCalculator` object.

A `FinancialsCalculator` object exposes the following properties:

Property	Type
Amount	CurrencyAmount

Property	Type
ReportingAmount	CurrencyAmount
TransactionIds	Key[]

To obtain the actual value of the calculation on this financial expression, use the `Amount` or `ReportingAmount` property on the `FinancialsCalculator` object, for example:

```
gw.api.financials.FinancialsCalculations.getTotalIncurredGross()
 .withClaim(claim)
 .Amount
```

**Note:** The `getTotalIncurredGross` method call returns a `FinancialsCalculator` object.

It is possible to link the `gw.api.financials.FinancialsCalculations` methods together to build an expression that returns the exact value that you want. All return a monetary value for the calculation on which they are called, limited by the passed-in parameters. The following examples illustrate this concept.

**Example 1.** The following example calculation returns the Open Reserves amount for the passed-in `Claim`.

```
gw.api.financials.FinancialsCalculations.getOpenReserves()
 .withClaim(clm)
 .Amount
```

**Example 2.** The following example calculation returns the amount of Open Reserves for the passed-in `Exposure` and `CostType` values only. It returns only those reserves and payments coded to this exposure and cost type, regardless of cost category coding.

```
gw.api.financials.FinancialsCalculations.getOpenReserves()
 .withExposure(exp)
 .withCostType(cType)
 .Amount
```

**Note:** Review the ClaimCenter GosuDoc on these methods for more details on how they work.

## Different Ways to Retrieve an Amount

Guidewire provides methods in the following classes that you can use to retrieve amounts associated with various financial calculations:

- `gw.api.financials.FinancialsCalculationUtil`
- `gw.api.financials.FinancialsCalculations`

### Retrieving Amounts Using the `FinancialsCalculationUtil` API

`FinancialsCalculationUtil.getAvailableReserves` returns a `FinancialsCalculator` object representing the Available Reserves. Use some variant of the following to retrieve the actual amount:

```
FinancialsCalculationUtil.getAvailableReserves().getAmount(claim, costCategory)
FinancialsCalculationUtil.getAvailableReserves().getAmount(claim, exposure, costType, costCategory)
...

```

Notice that the `getAmount` method filters the reserve lines that ClaimCenter includes in the calculation total.

### Retrieving Amounts Using the `FinancialsCalculations` API

`FinancialsCalculations.getAvailableReserves` returns a `FinancialsCalculator` object that also represents the Available Reserves. Use the following to retrieve the actual amount:

```
FinancialsCalculations.getAvailableReserves()
 .withClaim(claim)
 .withExposure(exposure)
 .withCostCategory(costCategory)
 .Amount
```

Notice that you must add individual filtering calls to the Gosu expression to filter the reserve lines that ClaimCenter includes in the calculation total.

# Predefined Financial Calculations

Guidewire provides calculations related to the following:

- “Payment Calculations” on page 623
- “Reserve Calculations” on page 624
- “Floating Financials Calculations” on page 625
- “Recovery Calculations” on page 625

## Payment Calculations

The following list describes the calculations related to payments that ClaimCenter provides in the base configuration.

---

### Calculated values

---

#### TotalPayments

Sum of all submitted and awaiting submission payments whose scheduled send date is today or earlier.

Access using:

```
FinancialsCalculations.getTotalPayments()
```

#### FuturePayments

Sum of all future payments (for example, those approved payments whose scheduled send date is after today). This includes both eroding and non-eroding payments.

Access using:

```
FinancialsCalculations.getFuturePayments()
```

#### PendingApprovalPayments

Sum of pending eroding and non-eroding payments.

Access using some variant of the following:

```
FinancialsCalculations.getFinancialsCalculation(FinancialsCalculationUtil.
PendingApprovalPaymentsExpression)
```

#### PendingApprovalErodingPayments

Pending approval eroding payments.

Access using some variant of the following:

```
FinancialsCalculations.getFinancialsCalculation(FinancialsCalculationUtil.
PendingApprovalErodingPaymentsExpression)
```

#### PendingApprovalNonErodingPayments

Pending approval non-eroding payments.

Access using some variant of the following:

```
FinancialsCalculations.getFinancialsCalculation(FinancialsCalculationUtil.
PendingApprovalNonErodingPaymentsExpression)
```

#### PaymentsForExAdjustments

Total foreign exchange adjustments for both eroding and non-eroding payments

#### ErodingPaymentsForExAdjustments

Total foreign exchange adjustment for *eroding* payments only.

Access using some variant of the following:

```
FinancialsCalculations.getFinancialsCalculation(FinancialsCalculationUtil.
ErodingPaymentsForeignExchangeAdjustmentsExpression)
```

#### NonErodingPaymentsForExAdjustments

Total foreign exchange adjustments for *non-eroding* payments only.

Access using some variant of the following:

```
FinancialsCalculations.getFinancialsCalculation(FinancialsCalculationUtil.
NonErodingPaymentsForeignExchangeAdjustmentsExpression)
```

## Reserve Calculations

The following list describes the calculations related to reserves that ClaimCenter provides in the base configuration.

---

### Calculated reserve values

---

#### TotalReserves

Sum of all submitted reserves and reserves that are awaiting submission.

Access using:

`FinancialsCalculations.getTotalReserves()`

#### AwaitingSubmissionReserves

Sum of all reserves that are awaiting submission.

Access using:

`FinancialsCalculations.getAwaitingSubmissionReserves()`

#### PendingApprovalReserves

Sum of all reserves that are pending approval.

Access using:

`FinancialsCalculations.getPendingApprovalReserves()`

#### OpenReserves

Total Reserves minus Total Payments

Detailed description: Sum of all submitted and awaiting submission reserves *minus* the sum of all submitted and awaiting submission eroding payments whose scheduled send date is today or earlier.

Access using:

`FinancialsCalculations.getOpenReserves()`

#### RemainingReserves

Open Reserves minus Future Eroding Payments

Detailed description: Similar to Open Reserves except that it includes future eroding payments. Thus, it is the sum of all submitted and awaiting submission reserves *minus* all approved eroding payments (awaiting submission eroding payments and future eroding payments).

This calculation can legitimately have a negative value. See “Predefined Reinsurance Calculations” on page 627 for an example.

Access using:

`FinancialsCalculations.getRemainingReserves()`

#### AvailableReserves

Remaining Reserve minus Pending Approval Eroding Payments

Similar to Open Reserves, except that it includes pending approval and future eroding payments. Thus, it is the sum of all submitted and awaiting submission reserves *minus* the sum of all approved eroding payments and all pending approval eroding payments.

This calculation can legitimately have a negative value. See “Predefined Reinsurance Calculations” on page 627 for an example.

Access using:

`FinancialsCalculations.getAvailableReserves()`

---

## Recovery Calculations

The following list describes the calculations related to recoveries that ClaimCenter provides in the base configuration.

---

### Calculated recovery values

---

**TotalRecoveries**

Sum of all submitted recoveries.

Access using:

```
FinancialsCalculations.getTotalRecoveries()
```

**TotalRecoveryReserves**

Sum of all submitted recovery reserves.

Access using some variant of the following:

```
FinancialsCalculations.getFinancialsCalculation(FinancialsCalculationUtil.
TotalRecoveryReservesExpression)
```

**TotalIncurredGross**

Open Reserves plus Total Payments, plus foreign exchange adjustments (described next in the detailed description), equivalent to Total Reserve plus Total NonEroding Payments plus foreign exchange adjustments described next.

Detailed description:

Sum of remaining reserves plus pending reserves plus noneroding payments plus awaiting submission noneroding payments plus foreign exchange adjustment for eroding payments plus foreign exchange adjustment for noneroding payments.

Access using some variant of the following:

```
FinancialsCalculations.getFinancialsCalculation(FinancialsCalculationUtil.
GrossTotalIncurredExpression)
```

**TotalIncurredNetRecoveries**

Gross Total Incurred value minus Total Recoveries.

Access using some variant of the following:

```
FinancialsCalculations.getFinancialsCalculation(FinancialsCalculationUtil.
TotalIncurredNetRecoveriesExpression)
```

**TotalIncurredNetRecoveryReserves**

Gross Total Incurred value minus Total Recovery Reserves.

Access using some variant of the following:

```
FinancialsCalculations.getFinancialsCalculation(FinancialsCalculationUtil.
TotalIncurredNetRecoveryReservesExpression)
```

## Floating Financials Calculations

The following list describes the calculations related to floating financials that ClaimCenter provides in the base configuration. Foreign exchange rates in ClaimCenter can be configured to be updated periodically. These calculations are used to update the *floating* financial components, such as reserve and recovery reserve amounts, using dynamic exchange rates. Payments and recoveries are not considered floating values and are evaluated with fixed rates from their respective transactions.

See “Using Floating Financial Calculations” on page 631.

**Note:** All the following calculations return a `FinancialsCalculator` object.

---

#### Calculated values

---

##### `AvailableReserves`

---

Remaining Reserves minus Pending Approval Eroding Payments

Sum of all submitted and awaiting submission reserves minus the sum of all approved, eroding payments and all eroding payments pending approval.

Access using:

```
FinancialsCalculations.getFloatingFinancials().AvailableReserves
```

##### `AwaitingSubmissionReserves`

---

Sum of all reserves that are awaiting submission today.

Access using:

```
FinancialsCalculations.getFloatingFinancials().AwaitingSubmissionReserves
```

##### `OpenRecoveryReserves`

---

Submitted recovery reserves minus submitted recoveries.

Access using:

```
FinancialsCalculations.getFloatingFinancials().OpenRecoveryReserves
```

##### `OpenReserves`

---

Total Reserves minus Total Payments

Sum of all submitted and awaiting submission reserves *minus* the sum of all submitted and awaiting submission eroding payments whose scheduled send date is today or earlier.

Access using:

```
FinancialsCalculations.getFloatingFinancials().OpenReserves
```

##### `PendingApprovalReserves`

---

Sum of all reserves that are pending approval.

Access using:

```
FinancialsCalculations.getFloatingFinancials().PendingApprovalReserves
```

##### `RemainingReserves`

---

Open Reserves minus Future Eroding Payments

Similar to Open Reserves, except that it includes future eroding payments. It is the sum of all submitted and awaiting submission reserves *minus* all approved, eroding payments (eroding payments awaiting submission and future eroding payments).

This calculation can legitimately have a negative value. See “Predefined Reinsurance Calculations” on page 627 for an example.

Access using:

```
FinancialsCalculations.getFloatingFinancials().RemainingReserves
```

##### `SubmittedOpenReserves`

---

Submitted, open reserves minus eroding, submitted payments.

Access using:

```
FinancialsCalculations.getFloatingFinancials().SubmittedOpenReserves
```

##### `TotalIncurredGross`

---

Sum of all open reserves and total payments.

Access using:

```
FinancialsCalculations.getFloatingFinancials().TotalIncurredGross
```

##### `TotalIncurredNet`

---

Sum of all open reserves and total payments minus total recoveries.

Access using:

```
FinancialsCalculations.getFloatingFinancials().TotalIncurredNet
```

---

**Calculated values**

---

**TotalIncurredNetMinusOpenRecoveryReserves**

Sum of all open reserves and total payments minus all open recovery reserves.

Access using:

`FinancialsCalculations.getFloatingFinancials().TotalIncurredNetMinusOpenRecoveryReserves`

**SubmittedTotalIncurredNet**

Sum of all open reserves

Reserves + Payments Submitted - Total Recoveries

Access using:

`FinancialsCalculations.getFloatingFinancials().SubmittedTotalIncurredNet`

---

## Predefined Reinsurance Calculations

Guidewire provides predefined calculations related to the subjects of the following topics:

- “Reinsurance” on page 627
- “Reinsurance Reserves” on page 628
- “Reinsurance Recoveries” on page 628

### Reinsurance

The following list describes the calculations related to reinsurance that ClaimCenter provides in the base configuration.

Calculation	Description
RITotalReinsurance	Sum of RI ceded reserves plus RI total recoverables. Access using: <code>FinancialsCalculations.getRITotalReinsurance()</code>
RINetNetPayments	Net payments (defined as total payments minus total recoveries) minus RI total recoverables. Access using: <code>FinancialsCalculations.getRINetNetPayments()</code>
RINetNetReserves	Open reserves minus RI open ceded reserves Access using: <code>FinancialsCalculations.getRINetNetReserves()</code>
RITotalIncurred	Sum of RI net payments plus RI net reserves Access using: <code>FinancialsCalculations.getRITotalIncurred()</code>
RISubmittedNetNetReserves	Submitted open reserves minus RI open ceded reserves. Access using: <code>FinancialsCalculations.getRISubmittedNetNetReserves()</code>
RISubmittedNetNetPayments	Submitted net payments (defined as total payments minus total recoveries) minus RI total recoverables. Access using: <code>FinancialsCalculations.getRISubmittedNetNetPayments()</code>
RISubmittedTotalIncurred	Sum of RI submitted net payments plus RI submitted net reserves. Access using: <code>FinancialsCalculations.getRISubmittedTotalIncurred()</code>

## Reinsurance Reserves

The following list describes the calculations related to reinsurance reserves that ClaimCenter provides in the base configuration.

Calculation	Description
RITotalCededReservesAdjustments	Committed ceded reserve adjustments. Access using: <code>FinancialsCalculations.getRITotalCededReservesAdjustments()</code>
RITotalCededReservesNonAdjustments	Committed ceded reserves non-adjusted. Access using: <code>FinancialsCalculations.getRITotalCededReservesNonAdjustments()</code>
RIOpenCededReserves	Total ceded reserves minus total RI recoverables. Access using: <code>FinancialsCalculations.getRIOpenCededReserves()</code>

## Reinsurance Recoveries

The following list describes the calculations related to reinsurance recoveries that ClaimCenter provides in the base configuration.

Calculation	Description
RITotalRecoverables	Total recoverables. Access using: <code>FinancialsCalculations.getRITotalRecoverables()</code>
RITotalRecoverablesAdjustments	Total committed recoverables including adjustments. Access using: <code>FinancialsCalculations.getRITotalRecoverablesAdjustments()</code>
RITotalRecoverablesNonAdjustments	Total committed recoverables not including adjustments. Access using: <code>FinancialsCalculations.getRITotalRecoverablesNonAdjustments()</code>

## Retrieving Transaction IDs

The `FinancialsCalculator.TransactionIds` property does not return the monetary amount of the calculation. Instead, it returns the IDs of the transactions that contribute to the calculated value. Thus, for example, executing the following returns the IDs of all transaction objects that contribute to the open reserves calculated values (reserves and payments).

```
gw.api.financials.FinancialsCalculations.getOpenReserves().TransactionIds
```

Again, it is possible to link multiple `FinancialsCalculator` methods together to uniquely identify the open reserves to include. For example:

- If you pass in only the claim, then the method returns the IDs of all the transactions that contribute to the Open Reserves for the entire claim.
 

```
gw.api.financials.FinancialsCalculations.getOpenReserves().withClaim(clm).TransactionIds
```
- If you use a version that takes an exposure only, then it returns the IDs of all the transactions that contribute to Open Reserves for that exposure *only*. It does not return the IDs of transactions for other exposures on the same claim.
 

```
gw.api.financials.FinancialsCalculations.getOpenReserves().withExposure(exp).TransactionIds
```

You can construct other versions of these methods that allow you to account only for certain Cost Types and/or Cost Categories, for example:

## Forming Composite Custom Expressions

Some of the `FinancialCalculationsUtil` methods end in `Expression`. These methods always return a `FinancialsExpression` object. You can combine the supplied `FinancialsExpression` instances to form composite custom expressions. For example, if ClaimCenter did not supply a `TotalIncurredNetRecoveryReserves` expression, you could create such an expression in either of the following ways:

```
//Example 1
gw.api.financials.FinancialsCalculationUtil.getTotalIncurredNetRecoveryReservesExpression()
 .minus(gw.api.financials.FinancialsCalculationUtil.getOpenRecoveryReservesExpression())

//Example 2
gw.api.financials.FinancialsCalculationUtil.getGrossTotalIncurredExpression()
 .minus(gw.api.financials.FinancialsCalculationUtil.getTotalRecoveryReservesExpression())
```

In either case, the method returns a new `FinancialsExpression` instance that is equivalent to the `TotalIncurredNetRecoveryReserves` expression.

However, to be useful, you need to obtain a calculated value for the financial expression. To do this, you must generate an instance of a `FinancialCalculator` object that is backed by this custom expression. This is done with a specialized method on `gw.api.financials.FinancialsCalculations` that returns the required `FinancialCalculator` object:

```
gw.api.financials.FinancialsCalculations.getFinancialsCalculation(FinancialExpression)
```

Thus, to generate a `FinancialCalculation` for the example `FinancialExpression`, you would use:

```
uses gw.api.financials.*

var TotalIncurredNetRecoveryReserves = FinancialsCalculations
 .getFinancialsCalculation(FinancialsCalculationUtil
 .getGrossTotalIncurredExpression()
 .minus(FinancialsCalculationUtil
 .getOpenRecoveryReservesExpression()))
```

## Creating Custom Financial Gosu Classes

Clearly, having to repeat this long string each time that you want to make use of a custom calculation is very tedious. It is also error prone and inefficient from a performance point of view. Instead, Guidewire recommends that you create a custom Gosu utility class that contains all your custom financial calculations, which insures that they are all defined in one location. You can then reference these calculations each place that you reference Gosu code (rules, Gosu classes and enhancements, and PCF files).

For example, to construct a custom version of `TotalIncurredGross`, you can do something similar to the following:

```
package util.financials;

uses gw.api.financials.FinancialsCalculation
uses gw.api.financials.FinancialsCalculationUtil

@Export
class CustomCalcs {
 construct() { }
 private static var calcMyTotalIncurredNet =
 FinancialsCalculationUtil.getFinancialsCalculation(FinancialsCalculationUtil
 .getGrossTotalIncurredExpression()
 .minus(FinancialsCalculationUtil.getTotalRecoveryReservesExpression()))

 public static property get MyTotalIncurredNet() : FinancialsCalculation {
 return calcMyTotalIncurredNet
 }
}
```

This Gosu class defines the custom calculation as a static property. As such, you need to access the property on the class itself, not an instance of the class. In other words, you do not need to construct an instance of the class using the new operator to access the property. In fact, any attempt to do so is a mistake.

You reference the example class by using the following syntax:

```
var amt = util.financials.CustomCalcs.MyTotalIncurredNet.getAmount(claim)
```

The base configuration sample rules show an example usage of the Guidewire-provided `util.financials.CustomCalcs` class in the Transaction Approval rule set. Since `MyTotalIncurredNet` is merely an instance of a `FinancialCalculator` object, the `getAmount` method works as described previously for the predefined calculations. Guidewire disables the sample rule that uses the `CustomCalcs` class in the base configuration.

#### Condition

```
TransactionSet.Subtype == "CheckSet"
```

#### Actions

```
var totalIncurredAmt = util.financials.CustomCalcs.MyTotalIncurredNet.getAmount(transactionSet.Claim)
var exceedAmount = gw.api.financials.CurrencyAmount.getStrict(20000, totalIncurredAmt.Currency)
if (totalIncurredAmt > exceedAmount) { transactionSet.requireApproval(
 displaykey.Rules.TransactionApproval.LimitExceeded.TotalIncurredOnClaimExceedsNC(
 gw.api.util.CurrencyUtil.renderAsCurrency(exceedAmount)))
}
```

## Financial Calculations with a Negative Value

Only two of the predefined financial calculations described at “Predefined Financial Calculations” on page 623 can legitimately have a negative value:

- Available Reserves
- Remaining Reserves

These two calculations are defined at “Reserve Calculations” on page 624. They can subtract payments for which offsetting reserves have not yet have been created.

The following example illustrates this concept.

Consider a claim or exposure with the following transactions:

- Submitted reserve: \$500.00
- Submitted payment: \$300.00
- Awaiting submission payment: \$400.00, scheduled send date is one week from the current day.

For this example to be possible, you need to set the `AllowPaymentsExceedReservesLimits` parameter in `config.xml` to `true` to allow payments to exceed reserves. With that parameter set, the Remaining Reserves in this case are -\$200.00. ClaimCenter does not automatically create the necessary offsetting reserves for the payment that exceeds reserves—the \$400.00 payment—until the scheduled send date of the payment check is reached. Because ClaimCenter counts these payments against remaining reserves, the Remaining Reserves value ends up negative.

Now, consider if an additional pending approval payment were created:

- Pending approval payment: \$350

In this case, the calculated values would be:

- Remaining reserves: -\$200.00
- Available reserves: -\$550.00

Available reserves are even further into negative territory, in this case because the pending approval payment is also subtracted.

## Eroding and Non-eroding Payments

It is important to understand the following about payments:

- *Partial* and *final* payments are eroding by default, which means they draw down the reserves for their reserve line. This can be changed by either of the following methods:
  - Setting the payment as non-eroding in the payment creation step of the New Check wizard (if this field is exposed)
  - Calling the `setAsNonEroding` domain method for a payment from either the Preupdate or Postsetup rules. (Once set as non-eroding, you can reset the payment to be eroding by using the `setAsEroding` domain method.)
- *Supplemental* payments are non-eroding by definition. Any partial or final payment can also be marked as non-eroding using one the previously described methods.

## Using Floating Financial Calculations

ClaimCenter includes support for the use of multicurrency reserving in financial transactions. When multicurrency reserving is disabled, although it is possible to create reserve transactions in currencies other than the claim currency, reserves are tracked and eroded in the claim's currency.

When multicurrency reserving is enabled, you can track and erode reserves and recovery reserves in the currency of choice. ClaimCenter provides the floating financials API to get financial calculation totals based on dynamic exchange rates. ClaimCenter includes a set of predefined floating financial calculations to support this feature. See “Floating Financials Calculations” on page 625.

This topic includes some examples that describe the use of floating financial calculations.

**Note:** It is assumed that multicurrency reserving is enabled in configuration for all of the following examples.

### Example 1. Using Reserve Lines in Multiple Currencies

In this example, the reserves and payments use currencies other than the claim currency.

1. Consider a claim filed on January 1, 2014 with a claim currency of Euros and a reporting currency of U. S. Dollars. A new reserve line is created with the reserve currency set to British pounds and a reserve amount of 100 pounds. The reserve is stored in the system in the transaction, reserving, claim, and reporting currencies, when the transaction is created.

Claim Currency—Euros (EUR)

Reporting Currency—U. S. Dollars (USD)

Reserve Line 1—100 pounds (GBP)

The exchange rates stored in ClaimCenter are:

GBP to EUR—1.2

EUR to USD—1.4

GBP to USD—1.6

**Note:** For every single transaction that occurs, ClaimCenter stores the following four amounts in the `TransactionLineItem` entity: `TransactionAmount`, `ReservingAmount`, `ClaimAmount`, and `ReportingAmount`. The transaction-to-claim, transaction-to-reserving, and claim-to-reporting exchange rates are stored in the corresponding fields in the `Transaction` entity.

2. On January 5, 2014, a payment of 10 pounds is created. The payment is stored in the system in the transaction, reserving, claim, and reporting currencies, when the transaction is created.

The exchange rates in ClaimCenter, dated January 1, 2014, are:

GBP to EUR—1.2

EUR to USD—1.4

GBP to USD—1.6

The available reserves are calculated as follows:

Financial Component	Transaction Amount	Reserving Amount	Claim Amount	Reporting Amount
Reserve	100 GBP	100 GBP	120 EUR	168 USD
Payment	10 GBP	10 GBP	12 EUR	16.8 USD
Available Reserves		100-10 = 90 GBP	120 -12 = 108 EUR	168 -16.8 = 151.2 USD

Available reserves are calculated as follows:

$$\text{Available Reserves} = \text{Reserves} - \text{Payments}$$

The values used in the equation are in the corresponding currencies—reserving, claim, or reporting. The calculations in this step access the `FinancialsCalculations` API, calling `FinancialsCalculations.getAvailableReserves()`, to arrive at these values.

3. The `FloatingFinancialsCalculations` API can be used when you need to calculate available reserves based on current market rates rather than rates that were in force when the reserves were created.

On February 1, 2014, assume the market exchange rates have changed, as follows.

GBP to EUR—1.1

EUR to USD—1.3

GBP to USD—1.45

ClaimCenter accesses the `FloatingFinancialsCalculations` API, calling `FinancialsCalculations.getFloatingFinancials().getAvailableReserves()`, with the following results.

Financial Component	Transaction Amount	Reserving Amount	Claim Amount	Reporting Amount
Reserve	100 GBP	100 GBP	120 EUR	168 USD
Payment	10 GBP	10 GBP	12 EUR	16.8 USD
Available Reserves		100-10 = 90 GBP	120 -12 = 108 EUR	168 -16.8 = 151.2 USD
Available Reserves (Floating)		100-10 = 90 GBP	(100 - 10) * 1.1 = 99 EUR	(100 - 10) * 1.45 = 130.5 USD

In the preceding table, the Available Reserves row shows the calculated amounts based on the exchange rates stored in the system when the transactions were created. The Available Reserves (Floating) row shows the calculated amounts with reserves updated to the latest exchange rates available in the system.

For example:

$$\text{Available Reserves (Floating) in Claim Currency} = (\text{Reserves} - \text{Payments}) \text{ in Reserve Currency} * \\ (\text{Exchange Rate to Claim Currency}) = (100 - 10) * (1.1) = 99 \text{ Euros}$$

In the Financials Summary screen, you can view amounts using market rates, that is, floating financial calculations. See “Financials Screens” on page 42 in the *Application Guide*.

4. On February 20, 2014, a recovery is received for the amount of 5 pounds.

The exchange rates in ClaimCenter, dated February 1, 2014, are:

GBP to EUR—1.1

EUR to USD—1.3

GBP to USD—1.45

Financial Component	Transaction Amount	Reserving Amount	Claim Amount	Reporting Amount
Reserve	100 GBP	100 GBP	120 EUR	168 USD
Payment	10 GBP	10 GBP	12 EUR	16.8 USD
Recovery	5 GBP	5 GBP	5.5 EUR	7.15 USD
Total Recoveries		5 GBP	5.5 EUR	7.15 USD
Total Incurred Net		100 - 5 = 95 GBP	120 - 5.5 = 114.5 EUR	168 - 7.15 = 160.85 USD
Total Incurred Net (Floating)		100 - 5 = 95 GBP	(100 - 10) * 1.1 + 12 - 5.5 = 105.5 EUR	(100 - 10) * 1.45 + 16.8 - 7.15 = 140.15 USD

The Total Incurred Net value is calculated as follows:

$$\text{Total Incurred Net} = (\text{Open Reserves} + \text{Total Payments}) - \text{Total Recoveries}$$

Open Reserves are calculated, in turn, as follows:

$$\text{Open Reserves} = \text{Total Reserves} - \text{Total Payments}$$

The Total Incurred Net row shows the calculated amounts based on the exchange rates stored in the system when the transactions were created. The Total Incurred Net (Floating) row shows the calculated amounts with reserves updated to the latest exchange rates available in the system.

For example:

$$\begin{aligned}\text{Total Incurred Net (Floating) in Reporting Currency} &= ((\text{Open Reserves in Reserve Currency} * \text{Exchange Rate to Reporting Currency}) + \text{Total Payments}) - \text{Total Recoveries} \\ &= ((100 - 10) * 1.45) + 16.8 - 7.15 \\ &= 140.15 \text{ USD}\end{aligned}$$

**Note:** Floating calculations are only applied to reserves and recovery reserves.

#### See also

- “Predefined Financial Calculations” on page 623.
- “Transactions” on page 288 in the *Application Guide*.
- “Multiple Currencies” on page 335 in the *Application Guide*.
- “Exchange Rates” on page 342 in the *Application Guide*.
- “Overview of the Financials Data Model” on page 612.



# Creating ClaimCenter Financial Transactions

This topic describes how to use the financial calculation APIs that create various kinds of transactions in ClaimCenter.

This topic includes:

- “Setting Reserves” on page 635
- “Creating Reserve Transactions Directly” on page 637
- “Creating Checks and Payments by Using CheckCreator” on page 637
- “Creating Recovery Transactions” on page 639
- “Creating Recovery Reserve Transactions” on page 640

**See also**

- “ClaimCenter Financial Calculations” on page 619

## Setting Reserves

You read (retrieve) the values for Available Reserves by using the following static method, which returns a `FinancialsCalculator` object:

```
gw.api.financials.FinancialsCalculations.getAvailableReserves
```

Available reserves are similar to open reserves except that they include pending approval and future eroding payments. Thus, they are the sum of all submitted and awaiting submission reserves minus the sum of all approved eroding payments and all pending approval eroding payments. For more information, see the definition of Available Reserves at “Reserve Calculations” on page 624.

### Setting Reserve Values

To set the level of AvailableReserves, you can use one of the methods described in the topics that follow.

You can also create a `ReserveSet` and reserve transactions directly. For details, see “Available Reserves Method on Reserve Line Objects” on page 636.

### `setAvailableReserves`

To set reserve values, you must call `setAvailableReserves` directly on one of the following objects:

- `Claim`
- `Exposure`

The following code shows the method parameters:

```
setAvailableReserves(costType, costCategory, amount, submittingUser)
```

The method sets the available reserves for an exposure or claim to the given amount by creating a new reserve transaction that increases or decreases the currently available reserves.

The method takes the following parameters:

Parameter	Description
<code>costType</code>	The cost type for the reserve. This value cannot be null, but it can be unspecified.
<code>costCategory</code>	The cost category for the reserve. This value cannot be null, but it can be unspecified.
<code>amount</code>	The amount to which to set the available reserves. The amount must be zero or greater, and cannot be negative.
<code>submittingUser</code>	User submitting this reserve.

Executing the `setAvailableReserves` method from a Gosu rule provides significantly different behavior than performing the equivalent action through the ClaimCenter interface. If you execute this method as part of a rule:

- The method creates a separate `ReserveSet` object.
- The method ignores any existing reserve set objects for the giving `ReserveLine` that have a status of Pending Approval.

### Available Reserves Method on Reserve Line Objects

ClaimCenter also provides a mirror to the method described previously for use with the `ReserveLine` object. For example:

```
reserveLine.setAvailableReserves("1000", User.util.CurrentUser)
```

This method sets the available reserves for this reserve line to the given amount by creating a new reserve transaction that increases or decreases the currently available reserves.

The method takes the following parameters:

Parameter	Description
<code>newReserveAmount</code>	The amount to which to set the new reserve. The amount must be non-null and zero or greater, and cannot be negative.
<code>submittingUser</code>	User submitting this recovery reserve.

#### See also

“Creating Reserve Transactions Directly” on page 637

## Creating Reserve Transactions Directly

ClaimCenter provides an API that you can use to create reserve transactions directly. This API is more flexible than those listed in “Setting Reserves” on page 635. The methods include the following:

- `Claim.newReserveSet()`
- `ReserveSet.newReserve( exposure, costType, costCategory )`
- `Reserve.addNewLineItem( amount, comments, lineCategory )`
- `ReserveSet.prepareForCommit()`

For example:

```
var claim : Claim
var exposure : Exposure
var costType : CostType
var costCategory : CostCategory

var reserveSet = claim.newReserveSet()
var reserve = reserveSet.newReserve(exposure, costType, costCategory)

// Modify the new Reserve transaction as you want – for example:
reserve.Currency = Currency.TC_EUR

// Set the amount to 100 EUR.
// A Reserve transaction must have only one TransactionLineItem,
// and its LineCategory must be null.
reserve.addNewLineItem(gw.api.financials.CurrencyAmount.getStrict(
 100, Currency.TC_EUR) , null, null)

// Add more reserve transactions if required

reserveSet.prepareForCommit()
```

The `newReserve` method returns the new transaction entity so that you can modify it. At minimum, you must call the `addNewLineItem` method to add a `TransactionLineItem` with the `Amount` of the transaction. You can also create a multicurrency Reserve transaction by modifying the `Currency` field on the transaction.

After you finish creating transactions, call the `ReserveSet.prepareForCommit` method to submit the reserve set for approval and to update the financial calculations. After calling the `prepareForCommit` method, it is not possible to modify the transaction amount, currency, exchange rates, or other key fields on the transaction.

## Creating Checks and Payments by Using CheckCreator

Guidewire provides methods on `gw.api.financials.CheckCreator` to use as you create new `Check` objects in Gosu business rules. This class provides a builder-like interface to assist you in creating `Check` entities and their subobjects.

You can use one of the following methods to create a `CheckCreator` object:

- `Claim.newCheckCreator`
- `Exposure.newCheckCreator`

Use the builder-like methods on `CheckCreator` to modify a new `Check` object before you submit the check for approval and before ClaimCenter updates any affected `TAccount` objects. For example, you can add additional payments or line items to the check before you submit the check.

The `CheckCreator` methods start typically with `with` and provide a way for you to specify parameters such as the cost type or the mail-to-address of the check. For example:

```
CheckCreator.withPayee(payee)
 .withPayeeRole(payeeRole)
 .withRecipient(recipient)
 .withMailToAddress(mailToAddress)
 .withReportabilityType(reportabilityType)
 .withCostType(costType)
 ...
```

The following example illustrates how to create a new CheckCreator object and pass in various parameters. You need, of course, to define or to retrieve values for each of the supplied parameters.

```
var claimCheck = claim.newCheckCreator()
 .withPayee(payee)
 .withPayeeRole(payeeRole)
 .withRecipient(recipient)
 .withMailToAddress(mailToAddress)
 .withReportabilityType(reportabilityType)
 .withCostType(costType)
 .withCostCategory(costCategory)
 .withPaymentType(paymentType)
 .withLineCategory(lineCategory)
 .withCheckAmount(checkAmount)
 .withComments(comments)
 .withMemo(memo)
 .withPayTo(payTo)
 .withScheduledSendDate(scheduledSendDate)
 .withRequestingUser(requestingUser)
```

In actual practice, you need supply only those parameters that meet your business needs. However, at a minimum, supply valid values for the following Check parameters. You cannot leave these values null.

- Payee
- PayeeRole
- CostType
- CostCategory
- PaymentType
- CheckAmount
- ScheduledSendDate

In a similar manner, you can create a check for an exposure. The following examples illustrate how to do this.

```
exposure.newCheckCreator().withPayee(payee)
 .withPayeeRole(payeeRole)
 .withRecipient(recipient)
 .withMailToAddress(mailToAddress)
 .withReportabilityType(reportabilityType)
 .withCostType(costType)
 .withCostCategory(costCategory)
 ...
```

After you create a CheckCreator object, you then use one of the following methods on CheckCreator to create the check:

Check creation methods	Description
createCheck	<p>Creates a check defined by the current state of CheckCreator. It is possible to modify the check and any subentities of the check after you create the check.</p> <p>The createCheck method call returns a new Check object. It also stores the new Check object inside the CheckCreator for the subsequent call to prepareForCommit. You must call CheckCreator.prepareForCommit to submit the check for approval and to update the appropriate TAccount objects.</p>
prepareForCommit	<p>Prepares the previously created check for commit to the database. Call this method after calling CheckCreator.createCheck and after performing any modifications to the check, such as adding additional Payments or Transaction Line Items. This methods submits the check for approval and updates the appropriate TAccount objects.</p> <p>It is not possible to modify the check after you call this method, except for extension fields.</p>
createAndPrepareForCommit	<p>Use to prepare and commit the check in one operation. This method calls the following methods in the listed order:</p> <ul style="list-style-type: none"> <li>• createCheck</li> <li>• prepareForCommit</li> </ul>

For example, on a specific exposure:

```
var cc = exp.newCheckCreator().withPayee(payee1)
 .withPayeeRole("checkpayee")
 .withCostType("claimcost")
```

```
.withCostCategory("autoglass")
.withPaymentType("final")
.withCheckAmount(100.00)
.withScheduledSendDate("2012-01-31" as java.util.Date)

var thisCheck = cc.createCheck()
thisCheck.BankAccountNumber = "123456789"
cc.prepareForCommit()
```

#### Notes

- A check with an eroding payment must have reserves already created, so it can erode the available reserves. This case is the typical one.
- You can create the check with a non-eroding payment, which does not require a reserve.
- There is also First and Final. In this case, the eroding payment is both the first and the final payment on a reserve line and does not need reserves, for simple claims like auto glass claims. An offset reserve is created automatically for a first and final payment.

For more information, see:

- “Setting Reserves” on page 635.
- The Gosu API reference for the latest information on Gosu types and methods. See “Gosu Generated Documentation (‘gosudoc’)” on page 37 in the *Gosu Reference Guide*.
- “Reserves” on page 290 in the *Application Guide*.
- “Payments” on page 296 in the *Application Guide*.

## Creating Recovery Transactions

Guidewire provides methods to create recoveries on the following objects:

- Claim
- Exposure
- RecoverySet

### Claim Objects

The following example illustrates how to create a recovery on a claim. You need to define or retrieve values for each of the supplied parameters. Also, you need to supply only those parameters that meet your business needs.

```
claim.createRecovery(payer,
costType,
costCategory,
recoveryCategory,
lineCategory,
recoveryAmount,
recoveryCurrency,
exchangeRateOverride,
customExchangeRateDescription,
claimAmountOverride,
reportingAmountOverride,
comments,
requestingUser)
```

### Exposure Objects

The following example illustrates how to create a recovery on an exposure.

```
exp.createRecovery(payer,
costType,
costCategory,
recoveryCategory,
lineCategory,
recoveryAmount,
comments,
requestingUser)
```

## Creating Recoveries Directly

ClaimCenter provides an API that you can use to create recoveries directly. This API is more flexible than those listed earlier in this topic. The methods include the following:

- `Claim.newRecoverySet()`
- `RecoverySet.newRecovery( exposure, costType, costCategory )`
- `RecoverySet.prepareForCommit()`

For example:

```
var claim : Claim
var exposure : Exposure
var costType : CostType
var costCategory : CostCategory

var recoverySet = claim.newRecoverySet()
var recovery = recoverySet.newRecovery(exposure, costType, costCategory)

// Modify the new Recovery transaction as you want - for example:
recovery.Currency = Currency.TC_EUR
recovery.RecoveryCategory = RecoveryCategory.TC_SALVAGE

// Set the amount to 100 EUR.
// A Recovery transaction must have at least one TransactionLineItem.
recovery.addNewLineItem(gw.api.financials.CurrencyAmount.getStrict(
 100, Currency.TC_EUR) , null, null)

// Add more recovery transactions if required

recoverySet.prepareForCommit()
```

The `newRecovery` method returns the new transaction entity so that you can modify it. At minimum, you must call the `addNewLineItem` method to add a `TransactionLineItem` with the `Amount` of the transaction. You can also create a multicurrency Recovery transaction by modifying the `Currency` field on the transaction.

After you finish creating transactions, call the `RecoverySet.prepareForCommit` method to submit the recovery set for approval and to update the financial calculations. After calling the `prepareForCommit` method, it is not possible to modify the transaction amount, currency, exchange rates, or other key fields on the transaction.

## Creating Recovery Reserve Transactions

You read (retrieve) the values for Open Recovery Reserves by using the following static method, which returns a `FinancialsCalculator` object:

```
gw.api.financials.FinancialsCalculations.getOpenRecoveryReserves
```

The Open Recovery Reserves calculation, Total Recovery Reserves minus Total Recoveries, is analogous to payments' decreasing open reserves. After a recovery is received, it decreases the open recovery reserve associated with that reserve line. For more information, see "Reserve Overview" on page 290 in the *Application Guide*.

## Setting Recovery Reserve Values

To set the level of Open Recovery Reserves, you can use one of the methods described in the topics that follow.

You can also create a recovery reserves set and recovery reserves transactions directly. For details, see "Recovery Reserve Method on Reserve Line Objects" on page 641.

### `setOpenRecoveryReserves`

To set recovery reserve values, you must call `setOpenRecoveryReserves` directly on one of the following:

- `Claim`
- `Exposure`

The following code shows the method parameters:

```
setOpenRecoveryReserves(costType, costCategory, recoveryCategory,
 newRecoveryReserveAmount, submittingUser)
```

The method sets the open recovery reserves for an exposure or claim to the given amount. It does so by creating a new recovery reserve transaction that increases or decreases the current open recovery reserves. The method takes the following parameters:

Parameter	Description
costType	The cost type for the reserve. This value cannot be null, but it can be unspecified.
costCategory	The cost category for the reserve. This value cannot be null, but it can be unspecified.
recoveryCategory	The recovery category for the recovery reserve. This value can be null.
newRecoveryReserveAmount	The amount to which to set the open recovery reserves. The amount must be non-null and zero or greater, and cannot be negative.
submittingUser	User submitting this recovery reserve.

## Recovery Reserve Method on Reserve Line Objects

ClaimCenter also provides a mirror to the method described previously for use with the `ReserveLine` object. For example:

```
reserveLine.setOpenRecoveryReserves("Salvage", "1000", User.util.CurrentUser)
```

This method sets the open recovery reserves for this reserve line to the given amount by creating a recovery reserve that increases or decreases the current open recovery reserves. It returns the new recovery reserve, or `null` if the recovery reserve is not created.

The method takes the following parameters:

Parameter	Description
recoveryCategory	The recovery category for the recovery reserve. This value can be null.
newRecoveryReserveAmount	The amount to which to set the open recovery reserves. The amount must be non-null and zero or greater, and cannot be negative
submittingUser	User submitting this recovery reserve.

## Creating Recovery Reserve Transactions Directly

ClaimCenter provides an API that you can use to create recovery reserve transactions directly. This API is more flexible than those listed previously in this topic. The methods include the following:

- `Claim.newRecoveryReserveSet()`
- `RecoveryReserveSet.newRecoveryReserve( exposure, costType, costCategory )`
- `RecoveryReserve.addNewLineItem( amount, comments, lineCategory )`
- `RecoveryReserveSet.prepareForCommit()`

For example:

```
var claim : Claim
var exposure : Exposure
var costType : CostType
var costCategory : CostCategory

var recoveryReserveSet = claim.newRecoveryReserveSet()
var recoveryReserve = recoveryReserveSet.newRecoveryReserve(exposure, costType, costCategory)

// Modify the new RecoveryReserve transaction as you want - for example:
recoveryReserve.Currency = Currency.TC_EUR

// Set the amount to 100 EUR.
// A RecoveryReserve transaction must have only one TransactionLineItem,
// and its LineCategory must be null.
recoveryReserve.addNewLineItem(gw.api.financials.CurrencyAmount.getStrict(
 100, Currency.TC_EUR) , null, null)
```

```
// Add more RecoveryReserve transactions if required
recoveryReserveSet.prepareForCommit()
```

The `newRecoveryReserve` method returns the new transaction entity so that you can modify it. At minimum, you must call the `addNewLineItem` method to add a `TransactionLineItem` with the `Amount` of the transaction. You can also create a multicurrency RecoveryReserve transaction by modifying the `Currency` field on the transaction.

After you finish creating transactions, call the `RecoveryReserveSet.prepareForCommit` method to submit the reserve set for approval and to update the financial calculations. After calling the `prepareForCommit` method, it is not possible to modify the transaction amount, currency, exchange rates, or other key fields on the transaction.

# Configuring ClaimCenter Financial Screens

This topic discusses how you can modify various ClaimCenter screens related to financials. It also discusses how to configure financial holds and bulk invoice payments.

This topic includes:

- “Configuring the Financial Summary Screen” on page 643
- “Configuring Reserve Behavior” on page 649
- “Configuring Checks and Payments” on page 654
- “Configuring the Check Wizard Recurrence Settings” on page 655
- “Configuring the Check Wizard’s Default Payment Type” on page 656
- “Configuring Financial Holds” on page 656
- “Configuring Bulk Invoice Payments” on page 658

## Configuring the Financial Summary Screen

This topic explains how to configure the claim **Financial Summary** page to reflect information in a manner consistent with your business practices. It covers the following:

- “The Financials Summary Page” on page 644
- “Configuring the Filter Drop-Down” on page 644
- “Defining the Model Used by a Panel Set” on page 646
- “Controlling the Display of the Financial Model” on page 647

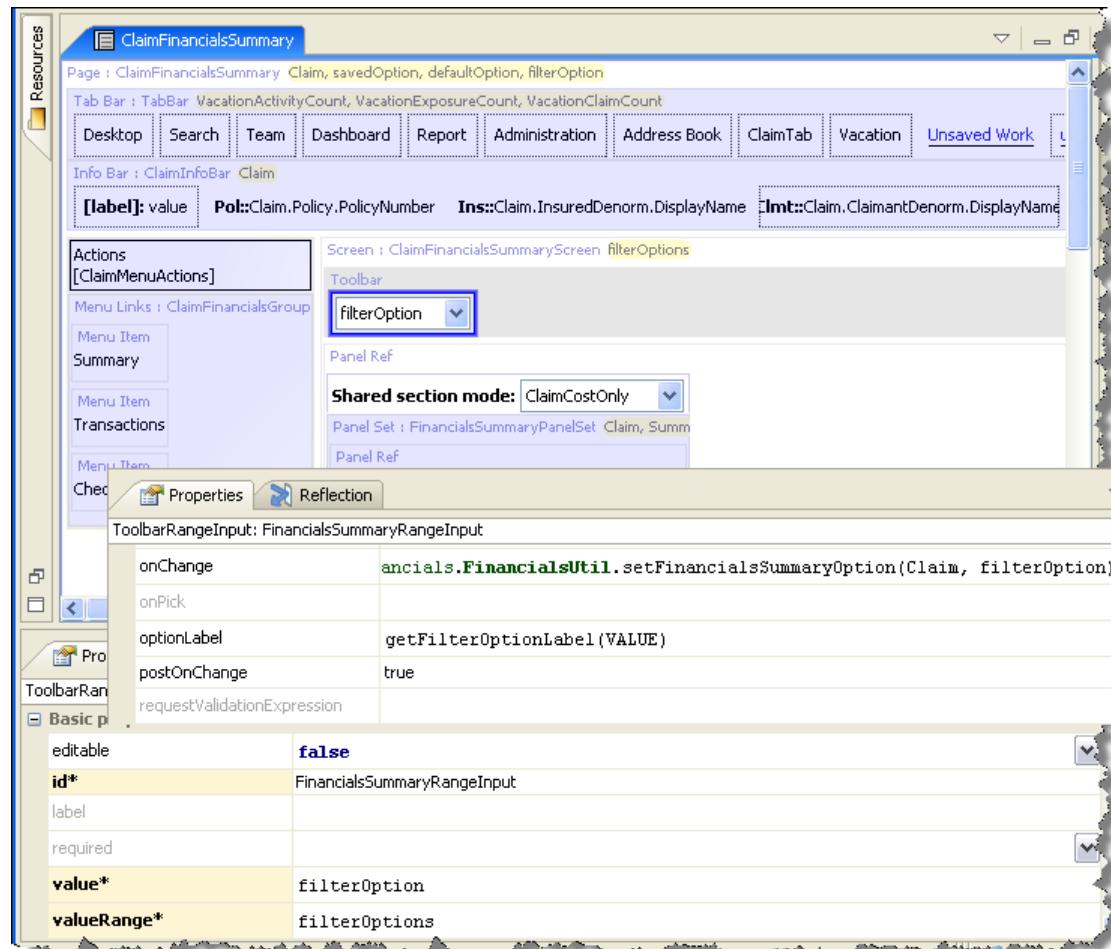
## The Financials Summary Page

You can configure the **Financials Summary** page to appear as you like. The page definition is made up files in the PCF  
→ claim → financials → summary folder in Studio:

ClaimFinancialsSummary	Defines the summary tab. You modify this page to change the contents of the drop-down that filters available options at the top of the screen.
FinancialsSummaryLV	Defines the list view that shows in the ClaimFinancialsSummary screen.
FinancialsSummaryPanelSet.Mode	Defines a model PCF that corresponds to an item in the drop-down filter. ClaimCenter renders a separate <i>Shared Section</i> panel (distinguished by <i>Mode</i> ) for each item in the filter. Base configuration modes include the following: <ul style="list-style-type: none"> <li>• Claimant</li> <li>• ClaimCostOnly</li> <li>• Coverage</li> <li>• Exposure</li> <li>• ExposureOnly</li> </ul>

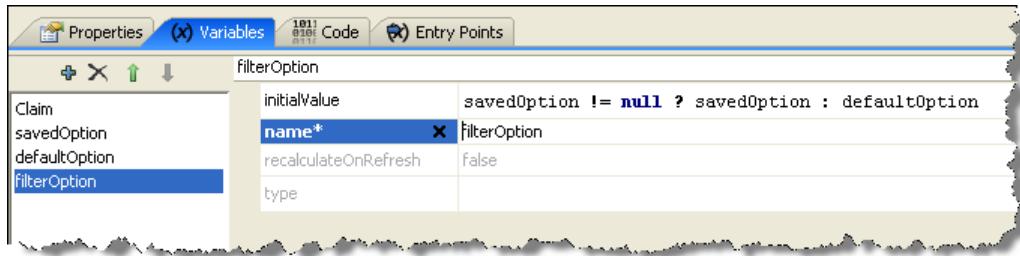
## Configuring the Filter Drop-Down

The **ClaimFinancialsSummary** file contains a **Toolbar** element that defines the filter list drop-down:



If you select the **Toolbar** element, you see a set of basic properties for it. You can also scroll down to see the list of advanced properties associated with it (the inset in the graphic).

Notice that the **value** property is set to `filterOption`. This is a page variable that you can access by selecting the entire page, then opening the **Variables** tab.



It sets the initial value to either the last saved option or to a default value (which, in this case, is **Exposure**).

Notice the **onChange** property is set to the following:

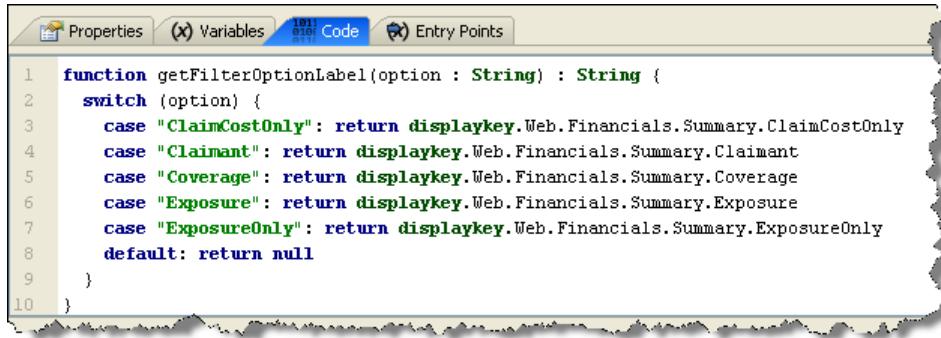
```
financials.FinancialsUtil.setFinancialsSummaryOption(Claim, filterOption)
```

This saves the filter option for the financials summary screen for the user session.

The **optionLabel** property uses a page method to determine the option label:

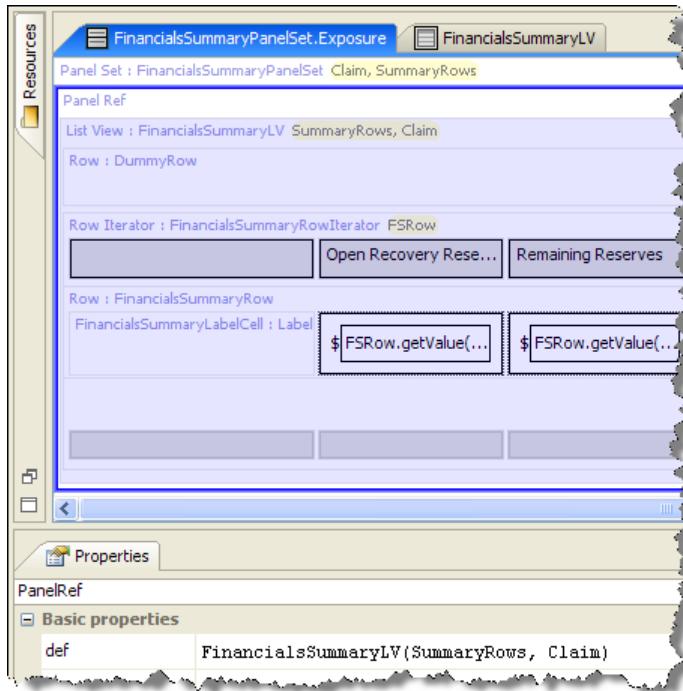
```
getFilterOptionLabel(VALUE)
```

You can access this method by selecting entire page, then opening the **Code** tab.



## Defining the Model Used by a Panel Set

For each item in the drop-down filter list, there is a panel set file. For example, the following graphic illustrates the Exposure mode of the `FinancialsSummaryPanelSet`. You edit this file to define the financial model used to determine both the levels and the columns that appear in your summary screens.



Notice that if you select **Panel Ref**, the **def** property is set to the following:

```
FinancialSummaryLV(SummaryRows, Claim)
```

The `SummaryRows` parameter is a page variable. Select that entire page, **Panel Set**, to see the **Variables** tab. This variable has the following definition:

```
(new financials.FinancialsSummaryModel(Claim,
 financials.FinancialsSummaryLevel1.EXPOSURE,
 financials.FinancialsSummaryLevel1.COSTTYPE,
 new financials.FinancialsExpression[] {
 gw.api.financials.FinancialsCalculationUtil.getRemainingReservesExpression(),
 gw.api.financials.FinancialsCalculationUtil.getFuturePaymentsExpression(),
 gw.api.financials.FinancialsCalculationUtil.getOpenRecoveryReservesExpression(),
 gw.api.financials.FinancialsCalculationUtil.getTotalPaymentsExpression(),
 gw.api.financials.FinancialsCalculationUtil.getTotalRecoveriesExpression(),
 gw.api.financials.FinancialsCalculationUtil.getTotalIncurredNetRecoveriesExpression()},
 false)).getFinancialsSummaryRows() as gw.api.financials.FinancialsSummaryRow[])
```

The `FinancialsSummaryModel` method constructor takes the following arguments:

<code>arg1</code>	Required. This is always set to <code>Claim</code> .
<code>arg2</code>	Required. The first level in an object summary.
<code>arg3</code>	Optional. The second level in the object summary.
<code>arg4</code>	A <code>FinancialExpression</code> array representing the column headings. You can specify the headings in any order.
<code>arg5</code>	A Boolean value indicating whether the panel is <i>claim cost</i> only.

You can specify any of the following values for `arg2` and `arg3` values:

- CLAIMANT
- EXPOSURE
- COVERAGE

- COSTTYPE
- COSTCATEGORY
- COSTTYPE\_COSTCATEGORY
- EXPOSURE\_COSTCATEGORY
- EXPOSURE\_COSTTYPE
- EXPOSURE\_COSTTYPE\_COSTCATEGORY

For the `FinancialExpression` array, you can use the `FinancialsCalculationUtil` API to retrieve any of the available financial building blocks. You can use these values alone or you can add or subtract them to calculate your own custom values. For example, as a financial expression, you can do the following:

```
gw.api.financials.FinancialsCalculationUtil.getFuturePaymentsExpression().
plus(gw.api.financials.FinancialsCalculationUtil.getTotalPaymentsExpression())
```

You can only use `.plus` or `.minus` to combine the financial building block values with the `FinancialsCalculationUtil` API.

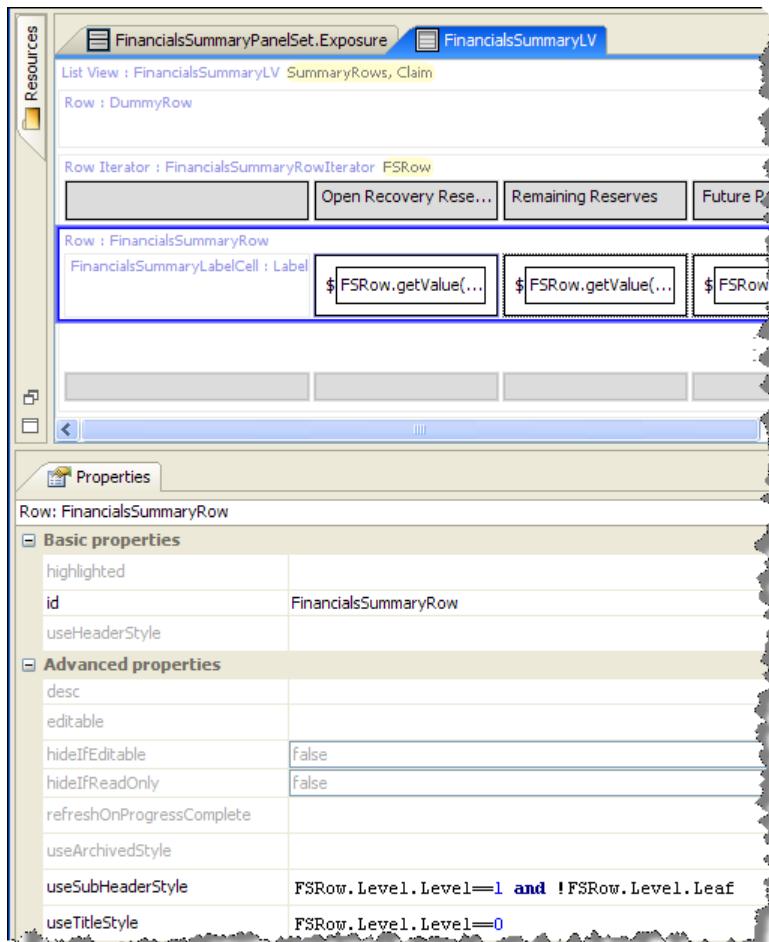
**See also**

- “ClaimCenter Financial Calculations” on page 619
- “Predefined Financial Calculations” on page 623

## Controlling the Display of the Financial Model

A list view controls how ClaimCenter displays the models defined in the `FinancialsSummaryPanelSet.Mode` file. By default, all the different modes use the same list view file, `FinancialSummaryLV`. The list view contains a

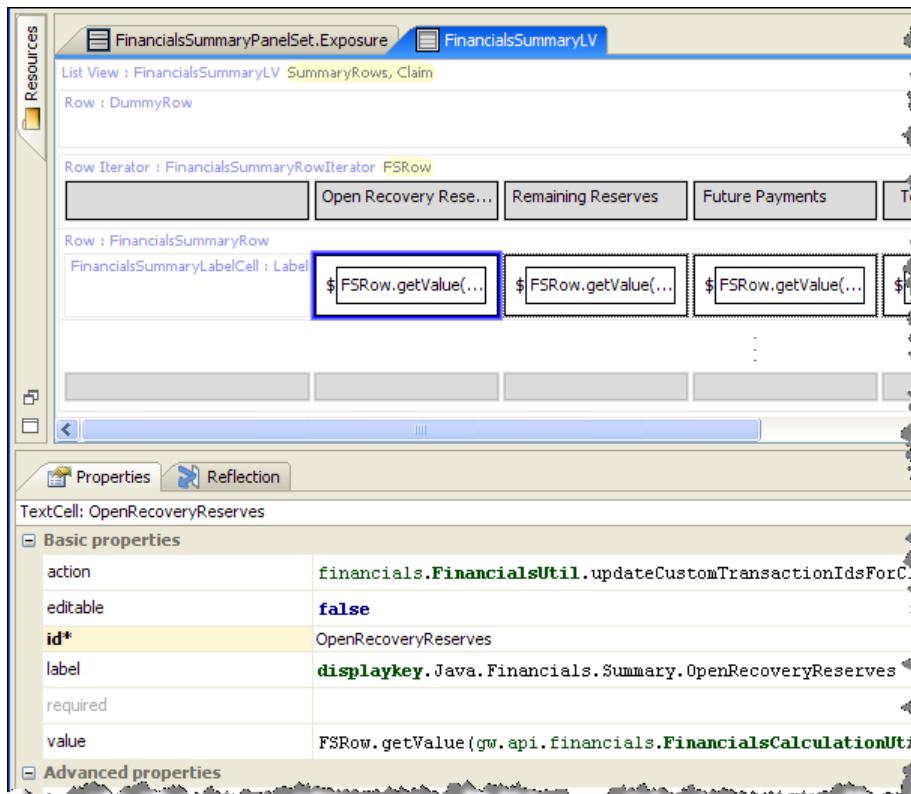
single `FinancialsSummaryRow` element containing multiple `FinancialSummaryCell` subelements. Each cell represents a column in the row.



The cell object has a `getValue` accessor that takes as an argument a `FinancialsExpression`. You can pass in any expression that you added to the model as you configured the `FinancialsSummaryPanelSet.Mode` file.

For example, if you try to access a value for an expression not in your `FinancialsSummaryPanelSet.Exposure` model, ClaimCenter throws an exception similar to the following:

`IllegalArgumentException: An expression, 'TotalPayments' was accessed in the FinancialsSummaryModel that was not specified on creation of the model (click for error details)`



If you want to change the list view, the interesting fields are `value` and `action`. These attributes must reference the same `FinancialExpression`. If the attributes do not match, a link for a value takes the user to a list of transactions that have nothing to do with the actual selected value.

You can control the visibility of an individual column by setting its permission in the `visible` attribute. The default list view hides columns that ClaimCenter does not permit users to see. For example, in the default configuration, the `FinancialSummaryCell:RemainingReserves` cell in the `FinancialsSummaryLV` PCF sets the following for the `visible` attribute:

```
perm.Claim.viewreserves(Claim) and perm.Claim.viewpayments(Claim)
```

This cell only displays the remaining reserves if the user has permission to view payments and reserves for a claim. For users without the permission, the Remaining Reserves column is invisible.

## Configuring Reserve Behavior

This topic describes how to configure the behavior of reserves and their related dialogs within ClaimCenter.

### Understanding How Configuration Impacts Reserves

The ClaimCenter Set Reserve screen has two mutually exclusive modes for setting reserves. The first mode is to set reserves by *available reserves* and second is to set reserves by *total incurred*. (You access the Set Reserves

screen by first opening a claim, then navigating the following path: Actions → New Transaction → Reserve.)

Set reserves by available reserves	It is possible to change the state of the reserves on a claim by changing the value of the <b>New Available Reserves</b> field for a reserve line.
Set reserves by total incurred	It is possible to change the state of the reserves on a claim by changing the value of the <b>New Total Incurred</b> field for a reserve line.

To set the mode to one or the other of these choices, edit the **SetReservesByTotalIncurred** parameter in the config.xml file.

- If the **SetReservesByTotalIncurred** value is **false**, ClaimCenter sets reserves by the available reserves.
- If the **SetReservesByTotalIncurred** value is **true**, ClaimCenter sets reserves by the total incurred.

The default is **false**.

During creation of a new reserve for a claim, ClaimCenter displays the current state of the entire claim's reserves including the available and unapproved reserves.

	Exposure	Coverage	*Cost Type	*Cost Category	Currently Available	Pending Approval	*New Available Reserves	Change	Comments
(1) 1st Party Vehicle - Ray Newton	Collision	Claim Cost	Auto body	\$400.00	-	\$	400.00		
(1) 1st Party Vehicle - Ray Newton	Collision	Expense - A&O	Other		-	\$	0		
(2) 1st Party Med Pay - Stan Newton	Medical payments	Claim Cost	Medical	\$2,000.00	-	\$	2000.00		
(2) 1st Party Med Pay - Stan Newton	Medical payments	Expense - A&O	Other		-	\$	0		
(3) 3rd Party Vehicle - Bo Simpson	Liability - Property damage	Claim Cost	Auto body	\$4,000.00	-	\$	4000.00		
Sum:				\$16,400.00	-	\$	0		

**Documents Linked to Reserves**

Remove	Name	View	Type	Status	Author	Date Modified
--------	------	------	------	--------	--------	---------------

Each line in this screen corresponds to one of the **ReserveLine** items on the claim. A **ReserveLine** is a unique combination of **Exposure**, **CostType**, and **CostCategory** on a particular claim. You can set reserves for, and make payments against, each reserve line item in a claim. The **SetReservesByTotalIncurred** parameter value sets which PCF page defines the reserve line items:

- If **SetReservesByTotalIncurred** is **true**, then the **EditableReservesLV.SetByNewTotalIncurred** PCF defines the content for the page.
- If **SetReservesByTotalIncurred** is **false**, then the **EditableReservesLV.SetByNewAvailableReserves** PCF defines the content for the page.

You can find these files in PCF → claim → newtransaction → reserve.

## The Fields in the Reserve Dialog

The following table describes the possible fields that can appear in the Set Reserve dialog:

Field	Description
Exposure	The claim's exposure corresponding to the reserve line. This is a read-only field for an existing ReserveLine.
Coverage	The policy coverage corresponding to the reserve. This is a read-only field.
Cost Type	The cost type corresponding to the reserve. This is a read-only field for an existing ReserveLine.
Cost Category	The cost category corresponding to the reserve. This is a read-only field for an existing ReserveLine.
Currently Available	The calculated amount of available reserves for this ReserveLine. This is a read-only field.
Pending Approval	The pending approval reserves for this reserve line. This field is read-only.
New Available Reserves	This field is visible only if SetReservesByTotalIncurred is false. If visible, you use this field to change reserves for a ReserveLine. The field is editable as long as the exposure for the specific line item is also open. <ul style="list-style-type: none"> <li>• If ClaimCenter displays the Set Reserve page, this field is equal to Available Reserves plus the Pending Reserves.</li> <li>• If a user enters a value in this field, the value represents the reserve the user wants for the ReserveLine.</li> <li>• After a user clicks Save, if the user requires approval for a reserve, ClaimCenter updates the Pending Approval field. Otherwise, it updates Available Reserves.</li> </ul>
Change	This field tracks the changes taking place between the time that a user first opens the Set Reserve dialog and the time that the user presses the Save button. The meaning of this field changes depending on how you have set SetReservesByTotalIncurred. <ul style="list-style-type: none"> <li>• If SetReservesByTotalIncurred is false, the field contains difference between the New Available Reserves and Available Reserves columns.</li> <li>• If SetReservesByTotalIncurred is true, the field contains the difference between the New Total Incurred and Total Incurred columns.</li> </ul>
Comments	Comments about the reserve. The value in this field is saved only for a new reserve or for a reserve for which the reserve amount changes.
New Total Incurred	This field is visible only if SetReservesByTotalIncurred is true. If visible, you use this field to change reserves for a ReserveLine. The field is editable as long as the exposure (or claim) for the specific reserve line is open as well. <ul style="list-style-type: none"> <li>• If ClaimCenter displays the Set Reserve page, this field is equal to Total Incurred plus the Pending Reserves.</li> <li>• If a user enters a value in this field, the value represents the reserve the user wants for the ReserveLine.</li> <li>• After a user clicks Save, if the user requires approval for a reserve, ClaimCenter updates the Pending Approval field. Otherwise, it updates Total Incurred.</li> </ul>
Total Incurred	This field is visible only if SetReservesByTotalIncurred is true. This field contains the current total incurred for this line item. This field is read-only.

## How Multiple Changes Interact

If a user changes the value in the Available Reserves on a particular reserve line to a new value and saves the change, ClaimCenter creates a new reserve transaction. (This works in a similar manner for the Total Incurred field, if you use total incurred.) The value of this transaction is the value of the change between the original Available Reserves value and the new value. (If using total incurred, then it is the value of the change between the Total Incurred field and the new value.)

If there are changes pending approval in the same reserve line, ClaimCenter does not add the new change to any existing **Pending Approval** values. Instead, ClaimCenter replaces the new changes appropriately. For example, suppose that you enter the **Set Reserves** dialog, which looks similar to the following:

Filtered by: A Reserve										
<a href="#">Exposure</a> ▾	<a href="#">Coverage</a> ▾	*Cost Type	*Cost Category	Currently Available	Pending Approval	*New Available Reserves	Change	Comments		
(1) 1st Party Vehicle - Ray Newton	Collision	Claim Cost	Auto body	\$400.00	-	\$400.00				
<b>Sum:</b>				<b>\$400.00</b>	-	<b>\$400.00</b>				

After you enter a new reserve value, ClaimCenter reflects the change:

Filtered by: A Reserve										
<a href="#">Exposure</a> ▾	<a href="#">Coverage</a> ▾	*Cost Type	*Cost Category	Currently Available	Pending Approval	*New Available Reserves	Change	Comments		
(1) 1st Party Vehicle - Ray Newton	Collision	Claim Cost	Auto body	\$400.00	-	\$700.00	\$300.00			
<b>Sum:</b>				<b>\$400.00</b>	-	<b>\$700.00</b>	<b>Get Sum</b>			

Then, suppose that you save the change, exit the dialog, and come back to it later. You see:

Filtered by: A Reserve										
<a href="#">Exposure</a> ▾	<a href="#">Coverage</a> ▾	*Cost Type	*Cost Category	Currently Available	Pending Approval	*New Available Reserves	Change	Comments		
(1) 1st Party Vehicle - Ray Newton	Collision	Claim Cost	Auto body	\$400.00	\$300.00	\$700.00				
<b>Sum:</b>				<b>\$400.00</b>	<b>\$300.00</b>	<b>\$700.00</b>				

Suppose that you continue to change the reserve even more:

Filtered by: A Reserve										
<a href="#">Exposure</a> ▾	<a href="#">Coverage</a> ▾	*Cost Type	*Cost Category	Currently Available	Pending Approval	*New Available Reserves	Change	Comments		
(1) 1st Party Vehicle - Ray Newton	Collision	Claim Cost	Auto body	\$400.00	\$300.00	\$900.00	\$500.00			
<b>Sum:</b>				<b>\$400.00</b>	<b>\$300.00</b>	<b>\$900.00</b>	<b>Get Sum</b>			

The next time you return to the dialog, if ClaimCenter has not yet approved the changes, you see the following:

Filtered by: A Reserve										
<a href="#">Exposure</a> ▾	<a href="#">Coverage</a> ▾	*Cost Type	*Cost Category	Currently Available	Pending Approval	*New Available Reserves	Change	Comments		
(1) 1st Party Vehicle - Ray Newton	Collision	Claim Cost	Auto body	\$400.00	\$500.00	\$900.00				
<b>Sum:</b>				<b>\$400.00</b>	<b>\$500.00</b>	<b>\$900.00</b>				

## Reserve Permissions and Authority Limits

You can use the following permissions to control access to reserve activities:

Permission	Description
claimviewrecres	View recovery reserves (and derived information) on a claim
claimviewres	View reserves (and derived information) on a claim
recrescreate	Create recovery reserve transactions
recresdelete	Delete recovery reserve transactions
recresedit	Edit recovery reserve transactions
rescreate	Create reserve transactions
resdelete	Delete reserve transactions
resedit	Edit reserve transactions

The following roles have all of the reserve-related permissions:

- Adjuster
- Claims Supervisor
- Clerical
- Manager
- New Loss Processing Supervisor
- Superuser

The following additional roles have only the `claimveiwrecres` and `claimviewres` permissions:

- Customer Service Representative
- Integration Admin
- Viewer

A user with the `rescreate` permission can create a reserve on an open claim provided the user has one of the proper `AuthorityLimitType` limits (defined in the `AuthorityLimitType` typelist). The following list describes the allowable *reserve* `AuthorityLimitType` values.

Code	Name	Description
car	Claim available reserves	The available reserves on or for a claim
ctr	Claim total reserves	The total reserves on or for a claim
ear	Exposure available reserves	The available reserves for a single exposure
etr	Exposure total reserves	The total reserves for a single exposure
rcs	Reserve change size	The size of a singe reserve change

A user can only create a reserve with a non-null exposure if the selected exposure is open.

## Setting the Number of Reserve Items to Show

You can change the number of reserve items that appear on the `Set Reserve` dialog. To configure this value, do the following:

1. Open `EditReservesLV`. You can find this files in PCF → claim → newtransaction → reserve.
2. Select the `RowIterator` widget near the top of the page. Scroll through the advanced options and find the `pageSize` attribute. The default value is 5.
3. Set the `pageSize` as needed.

4. Save and close the file.

## Configuring Checks and Payments

Each time a user creates a check, ClaimCenter associates it with at least one **Payment** transaction. This topic describes the points in ClaimCenter in which you can configure the behavior of the **Check** wizard and its associated payment fields.

### Understanding Checks and Payments

Each **Check** entity has one or more **Payment** entities associated with it. Each **Payment**, in turn, has one or more related **TransactionLineItem** objects that contain an **Amount** value. An individual **Payment** is *worth* the total of its **TransactionLineItem** amounts. The value of each **Check** is the sum of its related **Payment** objects.

The **PaymentType** typelist is a final typelist that sets a payment type. A payment can be a **partial**, a **final**, or a **supplement** type. Users set the check type on the **Check** wizard.

*Payments can either erode the reserves or not.* All supplemental payments do not erode reserves. A user must explicitly mark a payment that is **partial** or **final** as non-eroding. Otherwise, these payment types always erode reserves.

ClaimCenter maintains a running total of both the eroding and non-eroding payments. To access this information, use the following:

- `gw.api.financials.FinancialsCalculationUtil.getPendingApprovalErodingPaymentsExpression`
- `gw.api.financials.FinancialsCalculationUtil.getPendingApprovalNonErodingPaymentsExpression`

### Permissions and Authority Limits That Apply to Payments

The following are default system permissions associated with checks and payments:

Permission	Description
<code>claimviewpay</code>	View checks and payments (and derived information) on a claim
<code>clearedpayvoid</code>	Void a cleared check
<code>manpaycreate</code>	Create manual payment transactions
<code>manpaydelete</code>	Delete manual payment transactions
<code>manpayedit</code>	Edit manual payment transactions
<code>paycreate</code>	Create payment transactions
<code>paydelete</code>	Delete payment transactions
<code>payedit</code>	Edit payment transactions
<code>payrecode</code>	Recode a payment
<code>paystop</code>	Stop a check
<code>payvoid</code>	Void a check

In the base configuration, Guidewire grants the `clearedpayvoid` permission to the User Admin or Superuser roles only. Guidewire grants all other check and payment permissions to the following roles:

- Adjuster
- Claims Supervisor
- Clerical
- Manager
- New Loss Processing Supervisor

- Superuser

The following roles have the `claimviewpay` permission:

- Customer Service Representative
- Integration Admin
- Viewer

On the **Administration** page, you can set the following *payment* authority limits for users. (Again, these are taken from the `AuthorityLimitType` typelist.)

Code	Name	Description
cptd	Claim payments to date	The total amount of payments to date for the claim
eptd	Exposure payments to date	The total amount of payments to date for a single exposure
pa	Payment amount	The amount of a single payment
per	Payments exceed reserves	The amount by which payments are allowed to exceed reserves—if configuration parameter <code>AllowPaymentsExceedReservesLimits</code> is set to true. See “Application Configuration Parameters” on page 33 for more information on configuration parameters.

## Configuring the Check Wizard Recurrence Settings

Suppose that you would like to pre-populate recurrence settings in the Check wizard with values entered elsewhere in ClaimCenter. The recurrence settings appear in the **Set Check Instructions** step of the Check wizard.

To pre-populate these fields, navigate in Guidewire Studio to **Page Configuration (PCF)** → **claim** → **newtransaction** → **check** and click **NewPaymentInstructionsDV**. Locate the **CheckRecurrenceInputSet** widget and double-click the widget to open it in a separate view.

If you click the main **CheckRecurrenceInputSet** widget and then click the **Variables** tab below the widget, you see that there is `recurrenceHelper` variable defined. This variable maps to the following class:

```
gw.financials.CheckRecurrenceUIHelper
```

You can open this class and view or modify its business logic.

For example, to modify the defaults when creating a new weekly or monthly check recurrence, modify the `createRecurrenceWithDefaults` method on one of the inner classes `WeeklyRecurrenceUIHelper` or `MonthlyRecurrenceUIHelper`. These inner classes are defined at the bottom of `CheckRecurrenceUIHelper`.

## Check Recurrence Data Model

In the base configuration, ClaimCenter provides the following entities for working with check recurrence:

- Supertype `CheckRecurrence` contains the general information such as number of checks and first send date.
- Subtypes `MonthlyCheckRecurrence` and `WeeklyCheckRecurrence` provide more fields for granular frequency.

The following lists describe the important fields on these entities.

<b>CheckRecurrence</b>	
<code>IssuanceDateOffset</code>	Number of days before a check is due that ClaimCenter needs to issue the check
<code>FirstDueDate</code>	Due date of the first check in recurrence
<code>NumChecks</code>	Number of checks in the recurrence
<code>RecurrenceDay</code>	Day of the week the check is due

**CheckRecurrence****MonthlyCheckRecurrence**

MonthlyFrequency	Generate a check every <i>n</i> months
RecurrenceDate	Day of every month the check is due
RecurrenceWeek	Week in the month the check is due

**WeeklyCheckRecurrence**

WeeklyFrequency	Generate a check every <i>n</i> weeks
-----------------	---------------------------------------

## Configuring the Check Wizard's Default Payment Type

For a new payment on an open exposure, the **Payment Type** drop-down list has the following options:

- Partial
- Final

You can set the default for this control to one value or the other based on the selected **Cost Type** or some other value. First, edit the file at **Page Configuration (PCF) → claim → newtransaction → shared → NewPaymentDetailDV**. In this file, you create a Gosu method that sets the **Payment.PaymentType** property. You can create this method either in a Gosu class or in the PCF file. To create it in the PCF file, click the top level **DetailViewPanel NewPaymentDetailDV** and then click the **Code** tab below.

After defining the method, you can click the **Reserve Line** selector control on **NewPaymentDetailDV** and call it in the **onChange** property. In the base configuration, the **onChange** property sets the payment type to **null** with the code **Payment.PaymentType = null;**. If you change this code to use your method, the payment's payment type is updated when the user selects a particular reserve line.

## Configuring Financial Holds

You can configure several things about financial holds functionality, including:

- How ClaimCenter automatically sets the **Coverage in Question** field.  
See “Modifying the Automatic Setting of Coverage in Question” on page 656.
- The conditions that cause financial holds to be placed on a claim.  
See “Modifying the Conditions for Applying Financial Holds” on page 657.
- The conditions that permit **claimcost** initial reserves to be created.  
See “Modifying **claimcost** Initial Reserves” on page 657.

For an introduction to financial holds, see “Financial Holds” on page 325 in the *Application Guide*.

### Modifying the Automatic Setting of Coverage in Question

In the base ClaimCenter configuration, the **ClaimPreupdate** rule **CPU20000 - Coverage in question** and its children set the **Coverage in Question** field on the **Summary → Claim Status** screen. This rule calls a method found in a claim enhancement, **GWClaimFinancialHoldsEnhancement.gsx**.

The method called from CPU20000 is **claim.isCoverageInQuestion**. This method calls the helper method **getCoverageInQuestionFactors**, which does the real work to determine if any of the factors apply.

In the base configuration, the **GWClaimFinancialsHoldsEnhancement.getCoverageInQuestionFactors** method checks the following to see if the coverage is in question:

- If the loss date is after the policy expiration date
- If loss date is prior to the policy effective date

- If the policy status is something other than *In force* or *Archived*

Extending the `getCoverageInQuestionFactors` method to test for additional factors is as simple as adding an additional case, parallel to the other three. Check your new condition, and then add an entry to the return variable if your condition applies.

## Modifying the Conditions for Applying Financial Holds

In the base ClaimCenter configuration, ClaimCenter applies financial holds in the following cases:

- The claim's coverage is in question.
- The claim is marked as incident only.
- The claim's policy is unverified.

You can configure this functionality in Guidewire Studio in the transaction set validation rule **TXV15000 - Financial Holds** and its children. This functionality is controlled by the method `applyFinancialHolds` found in the `gw.entity.GWClaimFinancialHoldsEnhancement.gsx` claim enhancement.

### To add a new condition for applying financial holds

1. In ClaimCenter Studio, edit the `gw.entity.GWClaimFinancialHoldsEnhancement.applyFinancialHolds` method, adding an additional case for it to return `true`. The default method code is as follows:

```
function applyFinancialHolds() : Boolean {
 return this.CoverageInQuestion
 or this.IncidentReport
 or not this.Policy.Verified
}
```

2. Navigate in the resources pane on the left to **configuration → Rule Sets → Validation → TransactionSetValidationRules → Transaction Validation Rules → TXV15000 - Financial Holds**.
3. Add a new transaction validation rule under **TXV15000 - Financial Holds** that has specific warning and error rejections for the new case. Use one of the existing rules, such as **TXV15100 - Coverage In Question**, as a template.

## Modifying claimcost Initial Reserves

The base configuration of ClaimCenter prevents any `claimcost` initial reserves from being created when financial holds apply. This functionality is handled by checking for financial holds status before creating initial reserves in the **InitialReserve** rule set.

For example, navigate to **configuration → Rule Sets → InitialReserve → InitialReserve → Initial Reserve → IRR01000 -Auto → IRR01100 - Vehicle Damage → IRR01110 - Minor**. This rule checks for financial holds and creates a `claimcost` reserve only if there are no financial holds:

```
if(exposure.Claim.applyFinancialHolds() == true) {
 exposure.Claim.createNoteIfInitialReservesNotCreated()
} else {
 exposure.createInitialReserve(
 "claimcost", "body", ScriptParameters.InitialReserve_AutoMinorVehicleDamageBody)
}
```

---

**IMPORTANT** If you add rules to the **InitialReserve** rule set, you must also perform the check for financial holds before creating `claimcost` reserves. If you do not perform this check, ClaimCenter will create the reserves in the rule set and then reject them in the **TransactionSetValidationRules** rule set. Because the reserve and the new exposure are in the same bundle, you will not be able to save your new exposure.

## Configuring Bulk Invoice Payments

This topic provides information about the support ClaimCenter provides for paying bulk invoices and contains the following:

- “Overview of Bulk Invoices” on page 658
- “The Bulk Invoices Data Model” on page 659
- “Permissions for Bulk Invoice Processing” on page 659

### Overview of Bulk Invoices

You use a bulk invoice in Guidewire ClaimCenter to make a payment for an invoice that covers multiple claims. For example, this can be an insurance company that receives a single monthly invoice from a rental car company for all the rentals provided for the claims for that month. These types of invoices can have hundreds of line items (or more), all for different claims. Using Bulk Invoices, users can create a single payment that corresponds to the invoice and that assigns the appropriate portion of the payment to the individual claims.

ClaimCenter comes equipped with a set of permissions and roles that you can use to control access to the **Bulk Invoices** feature. These permission are separate from the standard payment permissions. A user with the proper permissions can work with the life cycle for a bulk invoice. This typically involves a user doing the following:

- Creating a bulk invoice object.
- Working on the invoice and saving a draft until all the associated information is complete.
- Validating the bulk invoice (and correcting any validation flags if necessary).
- Submitting the invoice for approval and subsequent payment.

ClaimCenter supports a batch process for processing bulk invoice payments. You can configure how often this batch process occurs. Guidewire also provides an **IBulkInvoiceAPI** that you can use to do post-processing on a bulk invoice payment. For example, you can detect holds or voids originating in an external application and have it reflected in the **Bulk Invoices** user interface. See the “Bulk Invoice Integration” on page 391 in the *Integration Guide* for information on working with this API.

The **Bulk Invoices** user interface allows users to work on the invoice over time. ClaimCenter saves a draft of the invoice until the user is ready to submit it for payment. Of course, before a user can submit a invoice, the invoice must be both validated and approved. A bulk invoice validation plugin is responsible for handling this validation. If you do *not* implement a validation plugin, after a user presses the **Validate** button, ClaimCenter simply marks the invoice as **Valid**. See “Bulk Invoice Integration” on page 391 in the *Integration Guide* for information on implementing a bulk invoice validation plugin.

If a bulk invoice requires approval, ClaimCenter creates an approval activity and assigns it to a user determined by the approval routing rules. You must write bulk invoice approval rule sets that manage approval routing. These rule sets are the only mechanism for controlling approval of bulk invoices. See “BulkInvoice Approval Rule Set Category” on page 45 in the *Rules Guide* for information on creating approval rule sets for bulk invoices.

Unlike other financial transactions, there are no specific authority limits for bulk invoices. Bulk invoices for payments are subject to the general financial transaction authority limits for that user. ClaimCenter checks each individual payment in the bulk invoice against the authority limit for the user.

## The Bulk Invoices Data Model

The following table lists the data entities associated with bulk invoices:

Entity	Description
BValidationAlert	An alert generated from the validation of a BulkInvoice. Your implementation of the IBulkInvoiceValidationPlugin is responsible for returning these objects as necessary. Each alert consists of a message and an alert type from the BValidationAlertType typelist.
BulkInvoice	Corresponds to the invoice requiring payment. This is the top-level entity. The creation and submission of a BulkInvoice entity results in a single large payment to the payee for this BulkInvoice. A BulkInvoice contains one or more BulkInvoiceItem objects.
BulkInvoiceItem	An individual line item on the bill that contains an amount and other fields necessary to code the cost of the item to a particular reserve line.
ReserveLineWrapper	Supports the creation of a draft reserve line while the BulkInvoice is in a draft state. This entity exists to support internal processing within ClaimCenter.

The following table lists the typelists associated with bulk invoices:

Typelist	Description
BValidationAlertType	Defines possible alerts returned from the validation plugin. This list contains a single alert type: <ul style="list-style-type: none"><li>• unspecified</li></ul> You can extend this list to support your validation plugin.
BulkInvoiceItemStatus	Status of a single BulkInvoiceItem. This value controls which actions are possible for a given Item. This list is final. You cannot extend it.
BulkInvoiceStatus	Defines business statuses of a BulkInvoice. This status controls which actions are possible for the invoice (such as edit, submit, void, and so forth). This list is final. You cannot extend it.

## Permissions for Bulk Invoice Processing

By default, the following system permissions control access to the Bulk Invoices functionality:

<code>bulkinvcreate</code>	Create a bulk invoice
<code>bulkinvdelete</code>	Delete a bulk invoice
<code>bulkinvedit</code>	Edit a bulk invoice
<code>bulkinvview</code>	View a bulk invoice

In the base application configuration, Guidewire grants these permissions, by default, to the following roles:

- Adjuster
- Claims Supervisor
- Clerical
- Customer Service Representative
- Manager
- New Loss Processing Supervisor
- Superuser

There are no specific authority limits that apply only to bulk invoices. See “The CheckAuthorityLimits Parameter and Bulk Invoices” on page 660 for more information on the interaction between system-defined authority limits and bulk invoices.

## Configuring the Bulk Invoices Feature

The `config.xml` file contains the following parameters that you can use to configure bulk payments:

<code>BulkInvoiceApprovalPattern</code>	The name of the activity pattern to use while creating bulk invoice approval activities. By default, the name of the activity is <code>approve_bulkinvoice</code> .
<code>AllowPaymentsExceedReservesLimits</code>	This parameter applies to all payments, not just bulk payments. If this value is <code>true</code> , bulk payments can exceed reserves for each <code>BulkInvoiceItem</code> on the invoice.

In the `scheduler-config.xml` file, you can configure how often the application runs the `bulkinvoiceesc` batch process. By default, this happens every day 15 minutes after midnight and 5:00 PM.

### The `CheckAuthorityLimits` Parameter and Bulk Invoices

Bulk invoices are not subject to the ClaimCenter standard payment authority limits. However, your bulk invoice configuration must take into account the `CheckAuthorityLimits` configuration parameter in `config.xml`. This is because ClaimCenter submits a check created for a bulk invoice item through the same approval process as it does for a check created through the **New Check Wizard**.

The `CheckAuthorityLimits` configuration parameter controls whether ClaimCenter checks authority limits for *all* financial transaction in ClaimCenter. By default, this parameter is `true` which means ClaimCenter *does* check. To check authority limits on most transactions but exclude bulk invoices, you can do the following:

- Set `CheckAuthorityLimits` to `false`.
- Use the `CheckSet.isForBulkedCheck` method to test whether a transaction set has an associated bulk invoice.
- Use the `TransactionSet.testAuthorityLimits` method in your approval rules to manually check authority limits for transactions not related to bulk invoices.

For more information about working with transaction approval in Gosu rules, see “Transaction Approval Rule Set Category” on page 73 in the *Rules Guide*.