

ClaimCenter Configuration Guide

Release 6.0.8



Copyright © 2001-2013 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 Live, 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.

This product includes information that is proprietary to Insurance Services Office, Inc (ISO). Where ISO participation is a prerequisite for use of the ISO product, use of the ISO product is limited to those jurisdictions and for those lines of insurance and services for which such customer is licensed by ISO.

This material is Guidewire proprietary and confidential. The contents of this material, including product architecture details and APIs, are considered confidential and are fully protected by customer licensing confidentiality agreements and signed Non-Disclosure Agreements (NDAs).

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

This product includes software developed by the Apache Software Foundation (<http://www.apache.org>).

Product Name: Guidewire ClaimCenter

Product Release: 6.0.8

Document Name: *ClaimCenter Configuration Guide*

Document Revision: 05-February-2013

Contents

About This Document.....	25
Intended Audience	25
Assumed Knowledge	25
Related Documents.....	25
Conventions In This Document	26
Support	26

Part I

ClaimCenter Configuration Basics

1 Overview of ClaimCenter Configuration.....	29
What You Can Configure	29
How You Configure ClaimCenter	30
Types of Application Environments.....	30
The Development Environment	30
The Production Environment	31
Deploying Configuration Files	31
Deploying Changes in a Development Environment	31
Deploying Changes to the Production Server.....	31
Regenerating the Data Dictionary and Security Dictionary	32
Managing Configuration Changes	33
Configuration Topics in This and Other Documents	33
2 Application Configuration Parameters	35
Working with Configuration Parameters	36
Accessing Configuration Parameters in Gosu.....	36
Configuration Parameter Attributes	37
Adding Custom MIME Types.....	38
Approval Parameters.....	38
BulkInvoiceApprovalPattern.....	38
PaymentApprovalPattern	38
RecoveryApprovalPattern	38
RecoveryReserveApprovalPattern	38
ReserveApprovalPattern	38
Archive Parameters.....	39
ArchiveEnabled.....	39
AssignClaimToRetriever	39
DaysClosedBeforeArchive	40
DaysRetrievedBeforeArchive	40
RestorePattern	40
SnapshotEncryptionUpgradeChunkSize.....	40
Assignment Parameters.....	40
AssignmentQueuesEnabled.....	40
Batch Process Parameters	41
BatchProcessHistoryPurgeDaysOld	41
BatchServer.....	41

Business Calendar Parameters	41
BusinessDayDemarcation	41
BusinessDayEnd	41
BusinessDayStart	41
BusinessWeekEnd	41
HolidayList (Obsolete)	41
IsFridayBusinessDay	42
IsMondayBusinessDay	42
IsSaturdayBusinessDay	42
IsSundayBusinessDay	42
IsThursdayBusinessDay	42
IsTuesdayBusinessDay	42
IsWednesdayBusinessDay	42
Cache Parameters	42
CacheActive	43
ExchangeRatesCacheRefreshIntervalSecs	43
GlobalCacheActiveTimeMinutes	43
GlobalCacheReapingTimeMinutes	43
GlobalCacheSizeMegabytes	43
GlobalCacheSizePercent	43
GlobalCacheStaleTimeMinutes	44
GlobalCacheStatsRetentionPeriodDays	44
GlobalCacheStatsWindowMinutes	44
GroupCacheRefreshIntervalSecs	44
ScriptParametersRefreshIntervalSecs	45
TreeViewRefresh	45
ZoneCacheRefreshIntervalSecs	45
Claim Catastrophe Parameters	45
MaxCatastropheClaimFinderSearchResults	45
Claim Health Indicator and Metric Parameters	45
ClaimHealthCalcMaxLossDateInYears	45
InitialReserveAllowedPeriod	45
MaxClaimResultsPerClaimHealthCalcBatch	45
Clustering Parameters	46
ClusteringEnabled	46
ClusterMulticastAddress	46
ClusterMulticastPort	46
ClusterMulticastTTL	46
ClusterProtocolStackOption1	46
ClusterProtocolStackOption2	47
ClusterProtocolStack	47
ConfigVerificationEnabled	47
PDFMergeHandlerLicenseKey	47
Database Parameters	48
QueryRewriteForClaimSearch	48
Deduction Parameters	49
BackupWithholdingTypeCode	49
CalculateBackupWithholdingDeduction	49
StandardWithholdingRate	49

Document Creation and Document Management Parameters	49
AllowActiveX	50
AllowActiveXAutoInstall	50
DisplayDocumentEditUploadButtons	50
DocumentContentDispositionMode	51
DocumentTemplateDescriptorXSDLocation	51
MaximumFileUploadSize	51
ReCallDocumentContentSourceAfterRollback	51
UseDocumentNameAsFileName	51
UseGuidewireActiveXControlToDisplayDocuments	52
Domain Graph Parameters	52
DomainGraphKnownLinksWithIssues	52
DomainGraphKnownUnreachableTables	52
Environment Parameters	52
AddressVerificationFailureAsError	52
BucketSizeForHistogramsOnAllIndexedColumns	53
CollectHistogramsOnAllIndexedColumns	53
CurrentEncryptionPlugin	53
DefaultApplicationLocale	53
DefaultCountryCode	53
DeprecatedEventGeneration	53
DiscardQueryPlansDuringStatsUpdateBatch	53
EnableAddressVerification	54
EnableInternalDebugTools	54
EntityValidationOrder	54
KeyGeneratorRangeSize	54
MemoryUsageMonitorIntervalMins	54
PublicIDPrefix	54
ResourcesMutable	55
RetainDebugInfo	55
StrictDataTypes	55
TwoDigitYearThreshold	56
UnreachableCodeDetection	56
UnrestrictedUserName	56
UseOldStylePreUpdate	56
WarnOnImplicitCoercion	56
WebResourcesDir	56
Financial Parameters	56
AllowMultipleLineItems	56
AllowMultiplePayments	56
AllowNegativeManualChecks	57
AllowNoPriorPaymentSupplement	57
AllowPaymentsExceedReservesLimits	57
CheckAuthorityLimits	57
CloseClaimAfterFinalPayment	57
CloseExposureAfterFinalPayment	57
DefaultApplicationCurrency	57
EnablePreSetupRulesInCheckCreator	58
Financials	58
MultiCurrencyDisplayMode	58
PaymentLogThreshold	59
SetReservesByTotalIncurred	59
UseDeductibleHandling	59
UseRecoveryReserves	59

Geocoding-related Parameters	59
ProximityRadiusSearchDefaultMaxResultCount	59
ProximitySearchOrdinalMaxDistance	59
UseGeocodingInAddressBook	60
UseGeocodingInPrimaryApp	60
UseMetricDistancesByDefault	60
Integration Parameters	60
ContactAutoSyncWorkItemChunkSize	60
EnableMetroIntegration	61
InstantaneousContactAutoSync	61
ISOPropertiesFileName	61
KeepCompletedMessagesForDays	61
LockPrimaryEntityDuringMessageHandling	61
MetroPropertiesFileName	61
PolicySystemURL	61
UseSafeBundleForWebServicesOperations	62
Miscellaneous Bulk Invoice Activity Pattern Parameters	62
BulkInvoiceItemValidationFailedPattern	62
BulkInvoiceUnableToStopPattern	62
BulkInvoiceUnableToVoidPattern	62
Miscellaneous Financial Activity Parameters	63
CheckDeniedPattern	63
CheckUnableToStopPattern	63
CheckUnableToVoidPattern	63
LastPaymentReminderPattern	63
RecoveryDeniedPattern	63
Miscellaneous Parameters	63
AllowBulkInvoiceItemsToHaveNegativeAmounts	63
AllowSoapWebServiceNamespaceCollisions	63
EnableClaimNumberGeneration	64
EnableClaimSnapshot	64
EnableStatCoding	64
LegacyExternalEntityArraySupport	64
ListViewPageSizeDefault	64
MaintainPolicyValidationLevelOnPolicyChange	65
MaxCachedClaimSnapshots	65
MaxStatCodesInDropdown	65
ProfilerDataPurgeDaysOld	65
StyleReportURL	65

PDF Print Settings Parameters	65
DefaultContentDispositionMode	65
PrintFontFamilyName	65
PrintFontSize	66
PrintFOPUserConfigFile	66
PrintHeaderFontSize	66
PrintLineHeight	66
PrintListViewBlockSize	66
PrintListViewFontSize	66
PrintMarginBottom	66
PrintMarginLeft	66
PrintMarginRight	67
PrintMarginTop	67
PrintMaxPDFInputFileSize	67
PrintPageHeight	67
PrintPageWidth	67
Scheduler and Workflow Parameters	67
IdleClaimThresholdDays	67
SchedulerEnabled	67
SeparateIdleClaimExceptionMonitor	68
WorkflowLogPurgeDaysOld	68
WorkflowPurgeDaysOld	68
WorkflowStatsIntervalMins	68
Search Parameters	68
DisableCBQTForClaimSearch	68
DisableCBQTForTeamGroupActivities	68
DisableHashJoinForClaimSearch	68
DisableHashJoinForProximitySearch	69
DisableIndexFastFullScanForClaimSearch	69
DisableIndexFastFullScanForProximitySearch	69
DisableIndexFastFullScanForRecoverySearch	69
DisableIndexFastFullScanForTeamGroupActivities	69
DisableSortMergeJoinForClaimSearch	69
DisableSortMergeJoinForTeamGroupActivities	69
MaxActivitySearchResults	70
MaxBulkInvoiceSearchResults	70
MaxCheckSearchResults	70
MaxClaimSearchResults	70
MaxContactSearchResults	70
MaxDocTemplateSearchResults	70
MaxDuplicateContactSearchResults	70
MaxNoteSearchResults	71
MaxPolicySearchResults	71
MaxRecoverySearchResults	71
SetSemiJoinNestedLoopsForClaimSearch	71

Security Parameters	71
EnableDownlinePermissions	71
FailedAttemptsBeforeLockout	71
LockoutPeriod	71
LoginRetryDelay	72
MaxACLParameters	72
MaxPasswordLength	72
MinPasswordLength	73
RestrictContactPotentialMatchToPermittedItems	73
RestrictSearchesToPermittedItems	73
ShouldSyncUserRolesInLDAP	73
UseACLPermissions	73
Segmentation Parameters	73
ClaimSegment	73
ClaimStrategy	73
ExposureSegment	73
ExposureStrategy	74
Spellcheck Parameters	74
CheckSpellingOnChange	74
CheckSpellingOnDemand	74
Statistics, Team, and Dashboard Parameters	74
AgingStatsFirstDivision	74
AgingStatsSecondDivision	74
AgingStatsThirdDivision	74
CalculateLitigatedClaimAgingStats	74
DashboardIncurredLimit	75
DashboardShowByCoverage	75
DashboardShowByLineOrLoss	75
DashboardWindowPeriod	75
GroupSummaryShowUserGlobalWorkloadStats	75
UserStatisticsWindowSize	75

User Interface Parameters	75
ActionsShortcut	75
AutoCompleteLimit	75
EnableClaimantCoverageUniquenessConstraint	76
HidePolicyObjectsWithNoCov...ragesForLossTypes	76
HighlyLinkedContactThreshold	76
IgnorePolicyTotalPropertiesValue	76
IgnorePolicyTotalVehiclesValue	76
InputHelpTextOnFocus	77
InputHelpTextOnMouseOver	77
InputMaskPlaceholderCharacter	77
ListViewPageSizeDefault	77
MaxBrowserHistoryItems (Obsolete)	77
MaxChooseByCoverageMenuItems	77
MaxChooseByCoverageTypeMenuItems	77
MaxClaimantsInClaimListViews	77
MaxTeamSummaryChartUserBars	78
QuickJumpShortcut	78
RequestReopenExplanationForTypes	78
ShowCurrentPolicyNumberInSelectPolicyDialog	78
ShowFixedExposuresInLossDetails	78
ShowNewExposureChooseByCoverageMenuForLossTypes	78
ShowNewExposureChooseByCoverageTypeMenuForLossTypes	78
ShowNewExposureMenuForLossTypes	79
UISkin	79
WizardNextShortcut	79
WizardPrevShortcut	79
WizardPrevNextButtonsVisible	79
Work Queue Parameters	79
InstrumentedWorkerInfoPurgeDaysOld	79
WorkItemCreationBatchSize	79
WorkItemRetryLimit	79
WorkQueueHistoryMaxDownload	80

Part II

The Guidewire Development Environment

3 Working with Guidewire Studio	83
What Is Guidewire Studio?	83
Starting Guidewire Studio	84
Restarting Studio	85
The Studio Development Environment	85
Working with the QuickStart Development Server	86
Connecting the Development Server to a Database	87
Deploying Your Configuration Changes	88
ClaimCenter Configuration Files	88
How ClaimCenter Interprets Modules	89
Key Directories	89
Studio File Structure	89
Recovering from Incorrect Data Model Changes	89
ClaimCenter Resources Tree	90
Configuring Diagnostic Logging in Studio	90
Configuring Guidewire Studio	91

Linking Studio to a SCM System	91
Configuring Version Control	94
Setting File Update and Deletion Parameters (General Settings Tab)	96
Setting Font Display Options	96
Setting Code Completion Options	97
Setting Server Default Options	99
Restarting the Application Server	100
Configuring External Editors	101
Associating an External Editor to a File Type	101
Working with an External XML Tool	102
Configuring the External Difference Tool	102
Setting the Studio Locale	103
4 ClaimCenter Studio and Gosu	105
Gosu Building Blocks	105
Gosu Case Sensitivity	106
Working with Gosu in ClaimCenter Studio	107
Gosu Packages	107
Gosu Classes	107
ClaimCenter Base Configuration Classes	108
Class Visibility in Studio	109
Preloading Gosu Classes	109
Gosu Enhancements	110
The Guidewire XML Model	111
Script Parameters	111
Working with Script Parameters	112
Referencing a Script Parameter in Gosu	113
5 Getting Started	115
Using the Studio Interface	115
Using the Studio Menus	116
Using the Toolbar Icons	117
Using the Right-Click Menu	118
Managing Windows and Views	118
Working with Studio Resources	120
Validating Studio Resources	121
6 Working in Guidewire Studio	123
Entering Valid Code	123
The Code Menu	124
Using Dot Completion	125
Using SmartHelp	126
Accessing the Gosu API Reference	127
Accessing the PCF Reference Guide	127
Accessing the Gosu Reference Guide	127
Using Studio Keyboard Shortcuts	128
Gosu Editor	129
PCF Editor	133
Viewing Keyboard Shortcuts in ClaimCenter	133
Using Text Editing Commands	134
Navigating Tables	134
Refactoring Gosu Code	134
Renaming a Gosu Resource	135
Moving a Gosu Resource	135
Saving Your Work	136

Part III Guidewire Studio Editors

7 Using the Studio Editors	139
Editing in Guidewire Studio	139
Working in the Gosu Editor	140
8 Using the Plugins Editor	141
What Are Plugins?	141
Plugin Classes	141
Startable Plugins	142
Working with Plugins	143
Customizing Plugin Functionality	144
9 Working with Web Services	145
What Are Web Services?	145
Using the Web Services Editor	145
Creating a New Web Service Proxy Endpoint	148
10 Implementing QuickJump Commands	149
What Is QuickJump?	149
Adding a QuickJump Navigation Command	150
Implementing QuickJumpCommandRef Commands	150
Implementing StaticNavigationCommandRef Commands	152
Implementing ContextualNavigationCommandRef Commands	152
Checking Permissions on QuickJump Navigation Commands	152
11 Using the Entity Names Editor	155
The Entity Names Editor	155
The Variable Table	156
The <i>Entity Path</i> Column	156
The <i>Use Entity Names?</i> Column	156
The <i>Sort</i> Columns	157
The Gosu Text Editor	157
Including Data from Subentities	158
Entity Name Types	159
12 Using the Messaging Editor	161
The Messaging Editor	161
Adding a Message Environment	161
Adding a Message Destination	162
Associating Event Names with a Message Destination	164
Working with Email Attachments	164
13 Using the Rules Editor	167
Working with Rules	167
Renaming or Deleting a Rule	169
Using Find-and-Replace	170
Changing the Root Entity of a Rule	171
Why Change a Root Entity?	172
Validating Rules and Gosu Code	173
Making a Rule Active or Inactive	173
14 Using the Display Keys Editor	175
The Display Keys Editor	175
Creating Display Keys in a Gosu Editor	176
Retrieving the Value of a Display Key	176

Part IV

Data Model Configuration

15 Working with the Data Dictionary	181
What is the Data Dictionary?	181
What Can You View in the Data Dictionary?	182
Using the Data Dictionary	183
Object Attributes.....	183
Entity Subtypes.....	184
Data Column and Field Types	184
Virtual Properties on Data Entities	185
16 The ClaimCenter Data Model	187
What Is the Data Model?	187
The Data Model in Guidewire Application Architecture	188
The Base Data Model	188
Using Dot Notation	188
Overview of Data Entities.....	189
Data Object Files.....	189
Working with Data Object Files	191
ClaimCenter Data Objects.....	192
Data Objects and the Application Database	193
Data Objects and Scriptability	195
Base ClaimCenter Data Objects	196
Component Data Objects	196
Delegate Data Objects.....	197
Delete Entity Data Objects	200
Entity Data Objects.....	200
Extension Data Objects.....	203
Nonpersistent Entity Data Objects	204
Subtype Data Objects	205
View Entity Data Objects	207
View Entity Extension Data Objects.....	208
Data Object Subelements	209
<array>	210
<column>	212
<componentref>.....	216
<edgeForeignKey>.....	217
<events>	220
<foreignkey>.....	220
<fulldescription>.....	223
<implementsEntity>	224
<implementsInterface>.....	225
<index>	226
<onetoone>	227
<remove-index>	228
<typekey>	229
17 Modifying the Base Data Model.....	233
Planning Changes to the Base Data Model.....	233
Overview of Data Model Extension	233
Strategies for Extending the Base Data Model	234
What Happens If You Change the Data Model?.....	235
Naming Guidelines for Extensions	236
Defining a New Data Entity	237

Extending a Base Configuration Entity	238
Working with Attribute Overrides	239
Extending the Base Data Model: Examples	241
Creating a New Delegate Object.....	241
Extending a Delegate Object.....	243
Defining a Subtype	246
Defining a Reference Entity	247
Defining an Entity Array.....	247
Implementing a Many-to-Many Relationship Between Entity Types	248
Extending an Existing View Entity.....	249
Removing Objects from the Base Configuration Data Model	250
Removing a Base Extension Entity.....	250
Removing an Extension to a Base Object.....	251
Implications of Modifying the Data Model	251
Deploying Data Model Changes to the Application Server	254
18 Working with Associative Arrays	255
Overview of Associative Arrays.....	255
Associative Array Types.....	256
Subtype Mapping Associative Arrays	256
Working with Array Values Using Subtype Mapping	257
Typelist Mapping Associative Arrays	258
Working with Array Values Using Typelist Mapping	259
19 Example: Creating Assignable Entities	263
Creating an Assignable Extension Entity	263
Step 1: Create Extension Entity UserAssignableEntityExt.....	263
Step 2: Create Assignment Class NewAssignableMethodsImpl	264
Step 3: Test Assign Your Extension Entity	266
Making Your Extension Entity Eligible for Round-Robin Assignment.....	267
Step 1: Extend the Assignment State Entities.....	267
Step 2: Subclass Class AssignmentType	268
Step 3: Create UserAssignableEntityExtEnhancement	269
Step 4: Test Round Robin Assignment.....	270
Creating an Assignable Extension Entity that Uses Foreign Keys.....	271
Step 1: Create Extension Entity TestClaimAssignable.....	271
Step 2: Add Foreign Keys to Assignable Entities.....	272
Step 3: Create New Assignment Type for New Extension Entity	272
Step 4: Create Enhancement TestClaimAssignableEnhancement	273
Step 5: Create Test Class TestClaimAssignableMethodsImpl	273
Step 6: Add Corresponding Permission for the Extension Entity.....	274
Step 7: Test Assignment of the Assignable Entity	276
20 The Domain Graph	279
What is the Domain Graph?	280
Understanding Graph Notation.....	280
Object Ownership and the Domain Graph	281
Accessing the Domain Graph	282
Viewing the Domain Graph	283
Adding Objects to the Domain Graph	284
Implementing the Correct Delegate	284
Defining Foreign Keys Between Objects	287
Graph Validation Checks	288
Working with Changes to the Data Model	289
Working with Shared Entity Data.....	290

Working with Cycles	290
21 Field Validation	293
Field Validators.....	293
Field Validator Definitions.....	294
<FieldValidators>.....	295
<ValidatorDef>.....	295
Modifying Field Validators	297
Using <columnOverride> to Modify Field Validation.....	297
Field Validation and Localization.....	298
22 Data Types.....	299
Overview of Data Types.....	299
Working with Data Types.....	300
Using Data Types	300
Defining a Data Type for a Property.....	301
The ‘datatypes.xml’ File.....	302
<...DataType>	302
Deploying Changes to datatypes.xml	302
Customizing Base Configuration Data Types	303
List of Customizable Data Types	303
Working with Money and Currency Data Types	304
Money Data Types	305
Currency Amount Data Types	307
Setting the Currency Display	308
Working with the Medium Text Data Type (Oracle).....	308
The Data Type API.....	308
Retrieving the Data Type for a Property.....	309
Retrieving a Particular Data Type in Gosu.....	309
Retrieving a Data Type Reflectively.....	309
Using the IDatatype Methods	309
Defining a New Data Type: Required Steps.....	310
Defining a New Tax Identification Number Data Type	310
Step 1: Register the Data Type	311
Step 2: Implement the IDatatypeDef Interface	311
Step 3: Implement the Data Type Aspect Handlers	312
23 Working with Typelists.....	315
What is a Typelist?	315
Terms Related to Typelists	316
Typelists and Typecodes.....	316
Typelist Definition Files	317
Different Kinds of Typelists	317
Internal Typelists.....	317
Extendable Typelists	318
Custom Typelists.....	318
Working with Typelists in Studio	318
The Typelists Editor	319
Entering Typecodes.....	321
Typekey Fields	322
Typelist Filters	325
Static Filters	325
Creating a Static Filter Using Categories	326
Creating a Static Filter Using Includes	328
Creating a Static Filter Using Excludes	329

Dynamic Filters	330
Category Typecode Filters	330
Category List Typecode Filters	330
Creating a Dynamic Filter	330
Mapping Typecodes to External System Codes	333

Part V User Interface Configuration

24 Using the PCF Editor	337
The Page Configuration (PCF) Editor	337
The PCF Canvas	338
Creating a New PCF File	339
The Toolbox Tab	340
The Structure Tab	340
The Translations Tab	340
The Properties Tab	341
Child Lists	342
PCF Elements	343
PCF Elements and the Properties Tab	343
Working with Elements	343
Adding an Element on the Canvas	344
Moving an Element on the Canvas	344
Changing the Type of an Element	345
Adding a Comment to an Element	345
Finding an Element on the Canvas	346
Viewing the Source of an Element	346
Duplicating an Element	346
Deleting an Element	346
Copying an Element	347
Cutting an Element	347
Pasting an Element	347
25 Introduction to Page Configuration	349
Page Configuration Files	349
Page Configuration Elements	350
Using Studio to Edit PCF Files	350
What is a PCF Element?	350
Types of PCF Elements	351
Identifying PCF Elements in the User Interface	353
Getting Started Configuring Pages	356
Finding an Existing Element To Edit	356
Creating a New Standalone PCF Element	357
26 Data Panels	359
Panel Overview	359
Detail View Panel	359
Define a Detail View	360
Add Columns to a Detail View	361
Format a Detail View	362
List View Panel	364
Define a List View	365
Iterate a List View Over a Data Set	367
Choose the Data Source for a List View	368

27 Location Groups	371
Location Group Overview	371
Define a Location Group	372
Location Groups as Navigation	373
Location Groups as Tab Menus	373
Location Groups as Menu Links	374
Location Groups as Screen Tabs	374
28 Navigation	377
Navigation Overview	377
Tab Bars	378
Configure the Default Tab Bar	379
Specify Which Tab Bar to Display	379
Define a Tab Bar	379
Define a Tab Bar Link	379
Define the Logout Link	380
Tabs	380
Define a Tab	380
Define a Drop-down Menu on a Tab	381
29 Configuring Spell Check	383
Using the Spell Checking Feature	383
Understanding How Spell Checking Works	383
How to Configure Spell Check	384
Step 1: Implement a Spell Check Utility	384
Step 2: Implement a <i>checkSpelling</i> Method	384
Step 3: Set Spelling Checker Parameters in <i>config.xml</i>	385
Step 4: Configure Text Fields for Spell Checking	385
30 Configuring Search Functionality	387
The ClaimCenter Search Functionality	387
File search-config.xml	388
The Criteria Definition Element	389
The Criterion Subelement	390
The Criterion Choice Subelement	392
Setting the Property Attribute to DateCriterionChoice	392
Setting the Property Attribute to FinancialCriterion	394
Setting the Property Attribute to ArchiveDateCriterionChoice	394
The Array Criterion Subelement	394
Deploying Customized Search Files	395
Steps in Customizing a Search	396
Working with Optional Search Criteria	396
Adding an Optional Search Field	396
Adding an Optional Array Search Field	398
31 Configuring Special Page Functions	401
Adding Print Capabilities	401
Overview of the Print Functionality	401
Types of Printing Configuration	403
Working with a <i>PrintOut</i> Page	404
Overriding the Print Settings in a File	407
Troubleshooting Print Configurations	407
Linking to a Specific Page: Using an <i>EntryPoint</i> PCF	407
Entry Points	408
Creating a Forwarding <i>EntryPoint</i> PCF	409

Linking to a Specific Page: Using an <i>ExitPoint</i> PCF	410
Creating an <i>ExitPoint</i> PCF	410
Populating a PCF Template: Using a <i>TemplatePage</i> PCF	412
A Simple Template Page	412

Part VI

Workflow and Activity Configuration

32 Using the Workflow Editor	419
Workflow in Guidewire ClaimCenter	419
Workflow in Guidewire Studio	419
Understanding Workflow Steps	421
Using the Workflow Right-Click Menu	422
Using Search with Workflow	422
33 Guidewire Workflow	423
Understanding Workflow	424
Workflow Instances	424
Work Items	425
Workflow Process Format	425
Workflow Step Summary	426
Workflow Gosu	426
Workflow Versioning	427
Workflow Localization	428
Workflow Structural Elements	428
<Context>	429
<Start>	429
<Finish>	429
Common Step Elements	429
Enter and Exit Scripts	429
Asserts	430
Events	430
Notifications	430
Branch IDs	431
Basic Workflow Steps	431
AutoStep	431
MessageStep	432
ActivityStep	433
ManualStep	434
Outcome	435
Step Branches	436
Working with Branch IDs	436
GO	437
TRIGGER	438
TIMEOUT	439
Creating New Workflows	440
Cloning an Existing Workflow	440
Extending an Existing Workflow	441
Extending a Workflow: A Simple Example	442
Instantiating a Workflow	444
A Simple Example of Instantiation	445
The Workflow Engine	446
Distributed Execution	447
Synchronicity, Transactions, and Errors	447

Workflow Subflows	450
Workflow Administration	450
Workflow Debugging and Logging	452
34 Defining Activity Patterns	453
What is an Activity Pattern?	453
Pattern Types and Categories	454
Activity Pattern Types	454
Categorizing Activity Patterns	455
Using Activity Patterns in Gosu	455
Calculating Activity Due Dates	455
Target Due Dates (Deadlines)	456
Escalation Dates	456
Defining the Business Calendar	456
Configuring Activity Patterns	457
Using Activity Patterns with Documents and Emails	459
Localizing Activity Patterns	459
Part VII	
Configuring Localization	
35 Localizing Guidewire ClaimCenter	463
Understanding Language and Locales	463
Working with Localization Configuration Files	464
36 Working with Locales	467
Locale Configuration Files	467
Adding a New Locale	468
Step 1: Add the Locale to the Localization File	468
Step 2: Add the Locale to the Language Type Typelist	472
Step 3: Add the Locale to the Collations File	472
Step 4: Create and Populate the New Locale Folder	473
37 Localizing the ClaimCenter Interface	475
Setting the Default Application Locale	475
Setting the User Locale	476
Configuring Zone Information	476
File <i>zone-config.xml</i>	477
Base Zone Hierarchies	479
Setting the IME Mode for Field Inputs	480
Printing in Non-US Character Sets	480
Configuring Apache FOP for Cyrillic	481
38 Localizing String Labels	483
Localizing Display Keys	483
Display Key Localization Files	484
Working with Missing Display Keys	484
Different Ways to Localize Display Keys	485
Localizing Typecodes	486
Accessing Localized Typekeys from Gosu	487
Exporting and Importing Localization Files	487
39 Localizing the Development Environment	491
Viewing Double-byte Characters in Studio	491
Changing the Studio Locale	492
Localizing Rule Set Names and Descriptions	492

Setting a Locale for a Gosu Code Block	493
Localizing Gosu Error Messages	494
Localizing Administration Tool Argument Descriptions	494
Localizing Logging Messages	496
40 Localizing Guidewire Workflow.....	497
Localize a Workflow	497
Setting a Locale for a Workflow	497
Localizing Workflow Step Names	498
Creating a Locale-Specific SubFlow	498
Localizing Gosu Code in a Workflow Step	499
41 Localizing Shared Administration Data	501
Shared Administration Data	501
Localized Entities	502
Localization Database Tables	503
Localized PCF Files	503
42 Localizing Field Validators.....	507
Localizing Field Validation	507
Using a Single Field Validators File	507
Using Multiple Field Validator Files	508
Configuring Localized Error Messages for Field Validators	508
Validating Country-Specific Entity Fields	509
Example Set Up	509
Step 1: Extend the PolicyAddress Entity	511
Step 2: Add the Input Field to the PolicyCenter Interface	512
Step 3: Add the Validator to the Default Field Validator File	513
Step 4: Create Country-Specific Validators and Error Messages	514
Step 5: Create the Validator Trigger Code	515
Testing the Field Validation Example	515
43 Localizing Templates	517
Understanding Templates: A Review	517
Creating Localized Documents, Emails, and Notes	518
Step 1: Create Locale-Specific Folders	518
Step 2: Localize Template Descriptor Files	519
Step 3: Localize Template Files	520
Step 4: Localize Documents, Emails, and Notes within ClaimCenter	521
Document Localization Support	522
IDocumentTemplateDescriptor	522
IDocumentTemplateSource	522
IDocumentTemplateSerializer	523
44 Localized Search and Sort.....	525
Searching and Sorting Character Data	525
How Guidewire Stores Data	526
Working with the Default Application Locale	526
Configuring a Locale	526
Database Searching	527
Searching and the Oracle Database	527
Searching and the SQL Server Database	530
Data Sorting	530
Sorting and the Oracle Database	530
Sorting and the SQL Server Database	531

45 Localizing Addresses	533
Address Localization Overview	533
The Address User Interface	534
<i>The AddressOwner and CCAddressOwner Interfaces</i>	534
The <i>AddressOwner</i> Interface	534
The <i>CCAddressOwner</i> Interface	534
<i>The CCAddressOwnerId and CountryAddressFields Classes</i>	535
Class <i>CountryAddressFields</i>	536
Class <i>CCAddressOwnerId</i>	536
PCF Address Configuration	537
46 Working with the Japanese Imperial Calendar	541
The Japanese Imperial Calendar Date Widget	541
Configuring Japanese Dates	542
Setting the Japanese Imperial Calendar as the Default for a Locale	543
Working with the Japanese Imperial Date Data Type	543
Setting a Field to Always Display the Japanese Imperial Calendar	544
Setting a Field to Conditionally Display the Japanese Imperial Calendar	545
Sample JIC Presentation Handler	546
Part VIII	
Testing Gosu Code	
47 Debugging and Testing Your Gosu Code	551
The Studio Debugger	551
Starting the Debugger	552
Setting Breakpoints	552
Minimizing the Impact to Users	553
Stepping Through Code	554
Viewing Current Values	554
Defining a Watch List	555
Resuming Execution	555
Using the Debugger with the GUnit Tester	555
Using the Gosu Tester	555
Testing a Gosu Expression	556
Testing a Gosu Program	556
Testing a Gosu Template	556
Testing Gosu Classes	557
Generating Testing and Debugging Information	557
Suggestions for Testing Rules	558
48 Using GUnit	559
The TestBase Class	559
Overriding TestBase Methods	560
Configuring the Server Environment	560
Configuring the Test Environment	563
Configuration Parameters	563
Creating a GUnit Test Class	565
Using Entity Builders to Create Test Data	568
Creating an Entity Builder	568
Entity Builder Examples	571
Creating New Builders	572
Running Gosu API Tests	577

Part IX Guidewire ClaimCenter Configuration

49 Using the Lines of Business Editor	583
Lines of Business in Guidewire ClaimCenter	583
The Studio Lines of Business Editor	584
The Context Menu	584
The Studio LOB Editor Tabs	586
Referential Integrity	586
Saving	587
Managing References to the LOB Typelists	587
Adding a New <i>LossType</i> Typecode	588
Adding a New <i>ExposureType</i> Typecode	589
50 Configuring Policy Behavior	591
Managing Aggregate Limits	591
Understanding Aggregate Limits	591
Define What Counts Against an Aggregate Limit	592
Configuring Policy Periods	593
Aggregate Limit Used Recalculation	595
Aggregate Limit Diagram	596
Specifying the Subtabs on a Policy	596
Defining Internal (ClaimCenter-only) Policy Fields	597
51 Configuring Financials	599
Understanding Financials	599
Double-entry Accounting	600
Overview of the Financials Data Model	600
Transaction States and the Financial Building Blocks	601
Files that Manage Financials	604
Configuration Parameters	604
Financial Summary Screen Configuration	605
The Financials Summary Page	605
Configuring the Drop-Down	606
Defining the Model Used by a Panel Set	607
Controlling the Display of the Financial Model	608
Configuring Reserve Behavior	610
Understanding How Configuration Impacts Reserves	610
Reserve Permissions and Authority Limits	614
Setting the Number of Reserve Items to Show	614
Checks and Payments Configuration	615
Understanding Checks and Payments	615
Permissions and Authority Limits That Apply to Payments	615
Batch Processes for Checks and Payments	616
Customizing the Check Wizard Recurrence Settings	619
Customizing the Check Wizard's Default Payment Type	619
Bulk Invoice Payment Configuration	619
Overview of Bulk Invoices	620
The Bulk Invoices Data Model	620
Configuring the Bulk Invoices Feature	621
52 Configuring Currency	623
Setting the Default Application Currency	623
Setting a Currency Mode	624
Working with Currency Typecodes	625

Implementing a Single Currency Configuration	625
Implementing a Multicurrency Configuration	626
Changing the Default Application Currency	626
53 Configuring Snapshot Views	629
How ClaimCenter Renders Claim Snapshots	629
Understanding Snapshot PCF Interaction	630
Encrypting Claim Snapshot Fields	630
Configuring Snapshot Templates	631
Schemas and Data Model Extensions	631
54 Configuring Deductibles	633
Deductible Data Model	634
Typekeys	635
Permissions	635
Deductibles and Checks	635
Transferring Checks	635
Recoding Payments	636
Deleting and Voiding/Stopping Checks	636
Denying or Resubmitting Checks	636
Applying Deductibles on Multicurrency Checks	636
Cleared or Issued Checks	636
Cloning Checks	636
Deductibles and Rules	637
55 Working with Catastrophe Bulk Associations	639
Catastrophe Bulk Association Configuration	639
The GWCatastropheEnhancement Class	640
Catastrophes Data Model	640
Catastrophe Configuration Parameter	640
56 Configuring Duplicate Claim and Check Searches	641
Understanding the Gosu Templates	641
Duplicate Claim Search	642
Duplicate Check Search	643
57 Configuring Claim Health Metrics	645
Adding a New Tier	645
Adding a High-Risk Indicator	647
Adding a New Claim Metric	649
Adding Your New Claim Metric or Indicator to Existing Claims	652
58 Configuring Recently Viewed Claims	655
Adding a Loss Date to the Recently Viewed Claim List	655
59 Configuring Incidents	659
Implicit Incidents	659
Explicit Incidents	659
To Create a New Incident Type	660
Quick Claim Configuration	660
Injured is a Contact Role	660
Incidents Data Model	660
Gosu and Incidents	660
Entities and Typelists Related to Incidents	662
60 Archiving Claims	665
Archive-related Documentation	665

Archiving and the Domain Graph.....	666
The ‘ClaimInfo’ Entity	667
Claims Archiving in Guidewire ClaimCenter	667
Archiving and Encryption.....	668
Selecting Claims for Archive Eligibility	668
Restoring Claims from the Command Line	669
Monitoring Claim Archiving Activity	669
Configuring Claims Archiving	670
Archiving-related Configuration Parameters	670
Archive Rules	671
Archive Events	672
Archive Work Queue	673
The Archive Plugin.....	673

About This Document

This document describes the different ways in which ClaimCenter can be configured, and describes the basic steps for implementing these configurations.

This topic includes:

- “Intended Audience” on page 25
- “Assumed Knowledge” on page 25
- “Related Documents” on page 25
- “Conventions In This Document” on page 26
- “Support” on page 26

Intended Audience

This document is intended for IT staff and system integrators who are configuring ClaimCenter either for an initial implementation or for ongoing enhancements.

Assumed Knowledge

This document assumes that you are familiar with running the server (basic system administration) and with using the ClaimCenter application (screens and functions).

Related Documents

See the following Guidewire documents for further information:

ClaimCenter Application Guide – Introduces the application, explains application concepts, and provides a high-level view of major features and business goals of ClaimCenter. This is your first place to look when learning about a feature. This book is written for all audiences.

ClaimCenter Installation Guide – Describes how to install a new copy of ClaimCenter into Windows or UNIX environments. This guide is intended for system administrators and developers who need to install ClaimCenter.

ClaimCenter System Administration Guide – Provides guidance for the ongoing management of a ClaimCenter system. This document is intended to help system administrators monitor ClaimCenter, manage its security, and take care of routine tasks such as system backups, logging, and importing files.

ClaimCenter Rules Guide – Describes the business rule methodology, rule categories for ClaimCenter, and rule syntax for Guidewire Studio. This book is intended for programmers who write Gosu business rules and analysts who define the business rule logic.

ClaimCenter SOAP API Reference Javadoc – Documents the SOAP APIs and entities for integration programmers. It includes: (1) Web service (SOAP) API interfaces; (2) ClaimCenter SOAP entities which are simplified versions of ClaimCenter entities; (3) SOAP-specific Java utility classes. See the *Integration Guide* for more

details.

ClaimCenter Gosu Generated Documentation – Documents all types visible from the Gosu type system. This includes Guidewire entities, Gosu classes, utility classes, Gosu plugin definitions, and Java types that are available from Gosu. With a local copy of the product, you can regenerate this documentation. From the `ClaimCenter/bin` directory run the `gwcc regen-gosudoc` command.

ClaimCenter Upgrade Guide – Provides instructions to upgrade ClaimCenter.

ClaimCenter Data Dictionary – Describes the ClaimCenter data model, including your custom data model extensions. To generate the dictionary, go to the `ClaimCenter/bin` directory and run the `gwcc regen-dictionary` command. To view the dictionary, open the `ClaimCenter/build/dictionary/data/index.html` file. For more information about generating and using the *Data Dictionary*, see the *ClaimCenter Configuration Guide*.

ClaimCenter Security Dictionary – Documents security permissions, roles, and the relationships between them. Generate the dictionary by going to the `ClaimCenter/bin` directory and running the `gwcc regen-dictionary` command. To view the dictionary, open the `ClaimCenter/build/dictionary/security/index.html` file. For more information about generating the *Security Dictionary*, see the *ClaimCenter Configuration Guide*.

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 .
monospaced	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.	Get the field from the <code>Address</code> object.
<i>monospaced italic</i>	Parameter names or other variable placeholder text within URLs or other code snippets.	<code>use 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 Center – <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 29
- “How You Configure ClaimCenter” on page 30
- “Types of Application Environments” on page 30
- “Deploying Configuration Files” on page 31
- “Regenerating the Data Dictionary and Security Dictionary” on page 32
- “Managing Configuration Changes” on page 33
- “Configuration Topics in This and Other Documents” on page 33

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 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 not changed during 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). For information on Guidewire Studio, see the “Using the Studio Editors” on page 139.

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 read-only directories for the base configuration application files.

- 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 a J2EE application server (IBM WebSphere, BEA WebLogic, or Apache Tomcat).
- 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. (See “Restarting the Application Server” on page 100 for information on restarting the application server after modifying configuration files.)

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 8 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 following commands from your *configuration* ClaimCenter root directory, based on the server type:

Server type	Command
Tomcat	gwcc build-war
WebLogic	gwcc build-weblogic-ear
WebSphere	gwcc build-websphere-ear

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 (by extending base entities, for example), then it is important that the *Data Dictionary* reflect those changes. In this way, other people who use the dictionary can see these new fields. To generate the *ClaimCenter Data Dictionary* and *ClaimCenter Security Dictionary*, run the following command from the `ClaimCenter/bin` directory:

```
gwcc regen-dictionary
```

This command generates the *ClaimCenter Data Dictionary* and *ClaimCenter Security Dictionary* in the following locations:

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

To view a dictionary, open its `index.html` file in the listed location.

Generating the dictionaries as you generate a .war (.ear) file. It is also possible to generate the data and security dictionaries as you generate the Guidewire application .war (.ear) file. To do so, use the following command:

```
gwcc build-war -Dconfig.war.dictionary=true
```

See “Commands Reference” on page 69 in the *Installation Guide* for information on the Guidewire command line tools for use in a development environment.

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/cc/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. For information on using Guidewire ClaimCenter with a SCM system, see “Linking Studio to a SCM System” on page 91.

Configuration Topics in This and Other Documents

Guidewire provides documents, such as the *ClaimCenter Rules Guide* and *ClaimCenter Integration Guide*, that cover specific customization topics. The following table provides a road map for more information regarding configuration topics:

Configuration area	Documentation
Organization	<i>ClaimCenter System Administration Guide</i>
Users and groups	
Coverage regions	
Security	<i>ClaimCenter System Administration Guide</i>
Roles	
Permissions	
Data visibility	
Financial authority limits	
System settings	<i>ClaimCenter Configuration Guide</i>
Configuration parameters	
Background process scheduler	
Claim information	
Typelists	
Data Model Extensions	
User interface	<i>ClaimCenter Configuration Guide</i>
Menus and toolbars	
Data views	
Changing text	
Client actions	
Activity patterns	<i>ClaimCenter Configuration Guide</i>

Configuration area	Documentation
Business rules	<i>ClaimCenter Rules Guide</i>
Integration	<i>ClaimCenter Integration Guide</i>
Programming interfaces	<i>ClaimCenter Rules Guide</i>
Document management integration	
Form letters	

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 all through the use of configuration parameters, along with much, much more.

This topic includes:

- “Working with Configuration Parameters” on page 36
- “Approval Parameters” on page 38
- “Archive Parameters” on page 39
- “Assignment Parameters” on page 40
- “Batch Process Parameters” on page 41
- “Business Calendar Parameters” on page 41
- “Cache Parameters” on page 42
- “Claim Catastrophe Parameters” on page 45
- “Claim Health Indicator and Metric Parameters” on page 45
- “Clustering Parameters” on page 46
- “Database Parameters” on page 48
- “Deduction Parameters” on page 49
- “Document Creation and Document Management Parameters” on page 49
- “Domain Graph Parameters” on page 52
- “Environment Parameters” on page 52
- “Financial Parameters” on page 56
- “Geocoding-related Parameters” on page 59
- “Integration Parameters” on page 60
- “Miscellaneous Bulk Invoice Activity Pattern Parameters” on page 62

- “Miscellaneous Financial Activity Parameters” on page 63
- “Miscellaneous Parameters” on page 63
- “PDF Print Settings Parameters” on page 65
- “Scheduler and Workflow Parameters” on page 67
- “Search Parameters” on page 68
- “Security Parameters” on page 71
- “Segmentation Parameters” on page 73
- “Spellcheck Parameters” on page 74
- “Statistics, Team, and Dashboard Parameters” on page 74
- “User Interface Parameters” on page 75
- “Work Queue Parameters” on page 79

Working with Configuration Parameters

You set application configuration parameters in file `config.xml`. You can find this file in the Guidewire Studio Resources tree 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. As Guidewire ClaimCenter maintains several copies of this file in different locations, always use Studio to modify configuration files as it manages the various copies for you.

WARNING Do not attempt to modify any files other than those in the `/modules/configuration` directory. Specifically, do **not** modify files in the `/modules/cc` directory. Any attempt to modify files in this directory can cause damage to the ClaimCenter application sufficient to prevent the application from starting thereafter.

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. If a parameter does not appear in the file, Guidewire ClaimCenter uses that parameter's default value (if it has one).

Note: ClaimCenter configuration parameters are case-insensitive.

IMPORTANT You cannot add new or additional configuration parameters. Guidewire does **not** support any attempt to do so.

Accessing Configuration Parameters in Gosu

To access a configuration parameter in Gosu code, use the following syntax, with XX being PL, BC, CC, or PC as appropriate.

```
gw.api.system.XXConfigParameters
```

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
- Can Change on Running Server
- 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 18 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.

Can Change on Running Server

Guidewire designates certain configuration parameters as `dynamic`. This indicates that Guidewire permits you to change this value on a running application server using JMX. The discussion of configuration parameters indicates this by adding `Can Change on Running Server: Yes` to the parameter description.

Note: For information on how to enable, configure, and monitor ClaimCenter using JMX, see “Enabling JMX with ClaimCenter” on page 91 in the *System Administration Guide*.

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 `<mimetypesmapping>` 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

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

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.

See also

- “Archiving” on page 87 in the *Application Guide* – information on archiving claims, searching for archived claims, and restoring archived claims.
- “Archive Parameters” on page 39 in the *Configuration Guide* – discussion on the configuration parameters used in claims archiving.
- “Archiving Claims” on page 665 in the *Configuration Guide* – information on configuring claims archiving, selecting claims for archiving, and archiving and the object (domain) graph.
- “Archiving Integration” on page 285 in the *Integration Guide* – describes the archiving integration flow, storage and retrieval integration, and the `IArchiveSource` plugin interface.
- “Archive” on page 48 in the *Rules Guide* – information on base configuration archive rules and their use in detecting archive events and managing the claims archive and restore process.
- “Logging Successfully Archived Claims” on page 37 in the *System Administration Guide*.
- “Purging Unwanted Claims” on page 58 in the *System Administration Guide*.
- “Archive Info” on page 160 in the *System Administration Guide*.
- “Upgrading Archived Entities” on page 62 in the *Upgrade Guide*.

IMPORTANT To increase performance, most customers find increased hardware more cost effective than archiving unless their volume exceeds one million claims or more. Guidewire strongly recommends that you contact Customer Support before implementing archiving to help your company with this analysis.

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 set to `true`, ClaimCenter creates a non-unique index on that column for `Extractable` entities. Furthermore, if the entity is `Keyable`, ClaimCenter creates a unique index on the `ID` and `ArchivePartition` columns.

IMPORTANT If you set `ArchiveEnabled` to `true`, the server refuses to start if you subsequently set it to `false`.

Default: `False`

Required: Yes

Permanent: Yes

AssignClaimToRetriever

Specifies to whom ClaimCenter assigns a restored claim:

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

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

Default: False

DaysClosedBeforeArchive

Used by the Claim Closed rule in the base configuration to set the DateEligibleForArchive field on Claim, which determines the date on which ClaimCenter archives a claim automatically. 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

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

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

Batch Process Parameters

BatchProcessHistoryPurgeDaysOld

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

Default: 45

BatchServer

Whether this server is the batch server. A value of `true` indicates that it is. A value of `null` is acceptable. Guidewire recommends that you **not** query on this parameter. Instead, use the following:

```
InitTab.getInstance().getEnvironment().isBatchServer()
```

Set for Environment: Yes

Can Change on Running Server: Yes

Business Calendar Parameters

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. 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

Cache Parameters

See also “Understanding Application Server Caching” on page 70 in the *System Administration Guide*.

CacheActive

Controls whether the application server cache is active. If you set this value to `false`, then you effectively set the maximum allowable space for the global cache to 0. In other words, if set to `false`, then ClaimCenter does not cache data at the application server level. See “Analyzing and Tuning the Application Server Cache” on page 73 in the *System Administration Guide* for more information.

Default: True

Set for Environment: Yes

ExchangeRatesCacheRefreshIntervalSecs

The time between refreshes of the `ExchangeRateSet` cache, in seconds. This integer value must be zero (0) or greater. See “Understanding Application Server Caching” on page 70 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 “Understanding Application Server Caching” on page 70 in the *System Administration Guide* for more information.

Default: 5

Minimum: 1

Maximum: 15

Set for Environment: Yes

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 “Understanding Application Server Caching” on page 70 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 “Understanding Application Server Caching” on page 70 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 “Understanding Application Server Caching” on page 70 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 “Understanding Application Server Caching” on page 70 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 “Understanding Application Server Caching” on page 70 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 “Understanding Application Server Caching” on page 70 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 “Understanding Application Server Caching” on page 70 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 “Understanding Application Server Caching” on page 70 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 “Understanding Application Server Caching” on page 70 in the *System Administration Guide* for more information.

Default: 86400

Claim Catastrophe Parameters

MaxCatastropheClaimFinderSearchResults

Maximum number of claims that ClaimCenter relates to a single catastrophe in the `CatClaimFinder` batch process, (used to find catastrophe-related claims). See “Batch Processes and Distributed Work Queues” on page 134 in the *System Administration Guide* for a discussion of ClaimCenter batch processes.

Default: 1000

Claim Health Indicator and Metric Parameters

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

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.

See Also

- “Managing Clustered Servers” on page 81 in the *System Administration Guide*

ClusteringEnabled

Whether to enable clustering on this application server.

Default: False

Set for Environment: Yes

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.

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.

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}):" +
"PING(timeout=2000;num_initial_members=4):" +
"MERGEFAST:" +
"FD(timeout=5000;max_tries=5;shun=true):" +
"VERIFY_SUSPECT(timeout=1500):" +
"pbcast.NAKACK(gc_lag=50;retransmit_timeout=600,1200,2400,4800):" +
"UNICAST(timeout=600,1200,2400,4800):" +
"pbcast.STABLE(desired_avg_gossip=20000):" +
"FRAG:" +
pbcast.GMS(join_timeout=3000;merge_timeout=10000;shun=true;print_local_addr=true));
```

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

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 44 in the *Installation Guide*
- “Setting Search Parameters for Oracle” on page 48 in the *System Administration Guide*

Deduction Parameters

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 an ActiveX control called Document Assistant to perform many of the functions related to document manipulation. ClaimCenter attempts to install this control whenever a client initiates a session. The

browser prompts the user to install the Guidewire Document Assistant. The Document Assistant ActiveX control is synonymous with the *TemplateRunner.ScriptControl* ActiveX control.

WARNING The Guidewire Document Assistant, like all ActiveX controls, has known security vulnerabilities. Contact Guidewire support for more information if you plan to use the Guidewire Document Assistant ActiveX control in production.

See Also

- “Configuring Guidewire Document Assistant” on page 25 in the *System Administration Guide*

AllowActiveX

Whether to allow ActiveX controls in the ClaimCenter interface (for document management, for example). 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`
- `UseGuidewireActiveXControlToDisplayDocuments`

See also `AllowActiveXAutoInstall`.

Default: True

AllowActiveXAutoInstall

Whether ClaimCenter automatically attempts to install the Guidewire Document Assistant and supporting JScript files.

If `AllowActiveXAutoInstall` is set to `false`, ClaimCenter does not attempt to install Guidewire Document Assistant and supporting JScript files. You must manually install the Guidewire Document Assistant and supporting JScript files if you want to use it. ClaimCenter does not attempt to download and install the control if it is not already present.

This configuration parameter works in conjunction with configuration parameter `AllowActiveX`:

- | | |
|--|---|
| <code>AllowActiveXAutoInstall = true</code> | <ul style="list-style-type: none">• If <code>AllowActiveX</code> = <code>true</code>, then ClaimCenter installs the ActiveX control and uses it.• If <code>AllowActiveX</code> = <code>false</code>, then ClaimCenter does not use the ActiveX control under any circumstances. |
| <code>AllowActiveXAutoInstall = false</code> | <ul style="list-style-type: none">• If <code>AllowActiveX</code> = <code>true</code>, then ClaimCenter does not install the ActiveX control. However, if you manually install the ActiveX control, then ClaimCenter does use the control.• If <code>AllowActiveX</code> = <code>false</code>, then ClaimCenter does not use the ActiveX control under any circumstances. |

Contact Guidewire Support for information about manually installing the control.

Default: True

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

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`

If ClaimCenter does not find the XSD file in that location, the application searches for the base version in the following location:

`ClaimCenter/modules/cc/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

ReCallDocumentContentSourceAfterRollback

Set this parameter to `true` to call the `isDocument` and `addDocument` methods again if there is a roll-back during a bundle commit. If a roll-back of the bundle does occur, then ClaimCenter rolls back each entity in the bundle.

The initial commit extracts the document content from the web request. This document content no longer exists after the bundle commit. Therefore, it is not possible to roll-back the document ID, even though it is possible to retrieve all other document fields. This can cause problems to occur.

Guidewire recommends the following:

- If you have extended the `Document` entity, then set this parameter to `true`.
- If you have **not** extended the `Document` entity, then it is most likely that you want to set this parameter to `false`.
- If you are using an `IDocumentMetadataSource`, then set the parameter to `false`. If you then experience validation exceptions, test your set-up, and, if necessary, set this value to `true`.

Default: `False`

UseDocumentNameAsFileName

Note: It is not necessary to set this configuration parameter in any locale that uses the English language. Use this configuration parameters with non-English locales only.

Configuration parameter `UseDocumentNameAsFileName` determines what you see as the document name for a document file opened through Guidewire ClaimCenter. This configuration parameter can take the following values:

- | | |
|--------------------|--|
| <code>False</code> | An opened document displays the document ID as its file name. |
| <code>True</code> | An opened document displays the actual document name as its file name. |

This is useful if you localize your environment and create document file names in the localization language. For example, suppose that you are working in the Russian locale. You create a document file and name it using Cyrillic characters:

- If you set `UseDocumentNameAsFileName` to `false`, then, as you open the document from within ClaimCenter, you see the opened document filename as the Document ID, for example, `123.jpg`.
- If you set `UseDocumentNameAsFileName` to `true`, then, as you open the document from within ClaimCenter, you see opened document filename as the Document Name, for example, `фото.jpg`. In other words, you see the Russian-language version of the document name.

Default: `False`

UseGuidewireActiveXControlToDisplayDocuments

Whether to use the Guidewire Document Assistant ActiveX control to display document contents. If `false`, ClaimCenter does not use the control and document contents return directly to the browser.

Default: `True`

Domain Graph Parameters

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 ensuring that 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

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`.

Default: `False`

BucketSizeForHistogramsOnAllIndexedColumns

Sets the bucket size to use for generating histograms on all indexed columns. Configuration parameter `CollectHistogramsOnAllIndexedColumns` must be true for this parameter to have any meaning.

Default: 254

CollectHistogramsOnAllIndexedColumns

Whether to collect histograms on all indexed columns or only on specific columns. This parameter is only applicable to Oracle databases.

Default: False

CurrentEncryptionPlugin

Set this value to the name of the plugin that you intend to use to manage encryption. See “Using the Plugins Editor” on page 141 for information on encryption plugins.

Default: IEncryption

DefaultApplicationLocale

Default localization locale for the application as a whole.

WARNING You set this parameter one-time only, before you start the application server for the first time. Any attempt to modify this parameter thereafter can invalidate your Guidewire installation.

Default: en_US

Set for Environment: Yes

Permanent: Yes

DefaultCountryCode

The default ISO country code to use if an address does not have a country explicitly set. ClaimCenter uses this also as the default for new addresses that it creates. See the *Guidewire Contact Management Guide* for more information on configuring addresses in Guidewire ClaimCenter.

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

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

DeprecatedEventGeneration

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

Default: False

DiscardQueryPlansDuringStatsUpdateBatch

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

Default: False

EnableAddressVerification

Set this value to `true` to enable address verification warnings.

Default: `False`

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 any entities not specified in this list after all listed entities.

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 field must be 9 characters or less in length. Use only letters and numbers.

Guidewire reserves for itself **all** public IDs that start with a two-character ID and then a colon, and thus reserves all `PublicIDPrefix` values with two characters.

IMPORTANT This field must be 9 characters or less in length. Use only letters and numbers. Do not use space characters, colon characters, or any other characters that might need to be processed or escaped specially by other applications. *Guidewire reserves all two-character-length values for this setting for current or future use.*

Required: Yes

Default: `None`

ResourcesMutable

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

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

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.

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

Default: `False`

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

StrictDataTypes = false. If you set this 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: False

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` (the default), 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

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

AllowNegativeManualChecks

Use to set the ability to create negative checks manually. The following values are valid:

- **True** – Allow user to create negative checks manually.
- **False** – Disable the ability to create negative checks manually.

The default ClaimCenter behavior is to not allow manual checks to be negative. If you need the ability to create negative checks manually, then you must explicitly set this parameter to **true**.

Default: False

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 limits for individual checks.

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 an exposure after the relevant final payment is made.

Default: True

DefaultApplicationCurrency

Default currency for the application as a whole. This is also sometimes known as *reporting currency*. You must set this parameter to a value contained in the *Currency* typelist. If using only a single currency for the application, then you set `MultiCurrencyDisplayMode` to **SINGLE**. In **SINGLE** mode, ClaimCenter uses the `CurrencyFormat` entries in each `GWLocale` in `localization.xml` to format the money amount, depending on the locale of each user.

WARNING You set this parameter one-time only, before you start the application server for the first time. Any attempt to modify this parameter thereafter can invalidate your Guidewire installation.

Default: usd

Set for Environment: Yes

Permanent: Yes

EnablePreSetupRulesInCheckCreator

It is often necessary to set additional values (used for transaction approval) on a check created in the **Auto First and Final** wizard. In most cases, ClaimCenter initializes these types of values in a Transaction Pre-setup rule. However, ClaimCenter does not execute the Transaction Pre-setup rule set for a claim created in the **Auto First and Final** wizard.

If you set the `EnablePreSetupRulesInCheckCreator` parameter to `true`, then ClaimCenter does invoke the Transaction Pre-setup rule set from the **Auto First and Final** wizard. Setting this parameter to `false` maintains the current behavior.

To summarize:

- | | |
|-------|--|
| False | The Auto First and Final FNOL wizard does not call the Transaction Pre-Setup rule set during check creation. |
| True | The Auto First and Final FNOL wizard does call the Transaction Pre-Setup rule set during check creation. |

Default: `False`

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) |
| 3 | financials subledger |

Default: `view`

MultiCurrencyDisplayMode

Sets whether Guidewire ClaimCenter displays money values in a single currency or in multiple currencies. Use one of the following:

- | | |
|----------|----------------------------------|
| SINGLE | Single currency mode |
| MULTIPLE | Always in multiple currency mode |

These values have the following meanings:

- | | |
|----------|--|
| SINGLE | In <code>SINGLE</code> mode, ClaimCenter assumes that all monetary amounts in the system are in the same currency. In this mode, ClaimCenter uses the <code>CurrencyFormat</code> entry in each <code>GWLocale</code> in <code>localization.xml</code> to format the money amount, depending on the locale of the user.

As all money values are in the same currency, you must ensure that the <code>CurrencyFormat</code> for each <code>GWLocale</code> specifies the money format for that one currency. It is possible to set slightly different formatting based on local custom, but all money formatting must be for the one default currency.

Guidewire strongly recommends that you set configuration parameter <code>DefaultApplicationCurrency</code> to its correct value if you set <code>MultiCurrencyDisplayMode</code> to <code>SINGLE</code> . This ensures the correctness of the data in the database if you upgrade to multicurrency at a later time. |
| MULTIPLE | In <code>MULTIPLE</code> mode, ClaimCenter obtains money formatting information from <code>currencies.xml</code> . In this mode, ClaimCenter ignores any <code>CurrencyFormat</code> information in <code>localization.xml</code> . However, even though unused, a tag for the default currency must be present in file <code>localization.xml</code> , even in <code>MULTIPLE</code> mode. |

For more information, see “Working with Locales” on page 467. See also “DefaultApplicationCurrency” on page 57.

Default: SINGLE

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

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. (See “Deductible Handling” on page 183 in the *Application Guide*.) If this value is `true`, then `AllowMultipleLineItems` must be `true` as well.

Default: True

UseRecoveryReserves

Whether to use recovery reserve transactions in financial calculations.

Default: True

ZeroReservesLevel

Specifies the level at which to zero-out reserves. Specify either 0 (cost type) or 1 (cost category).

Default: 0

Minimum: 0

Maximum: 1

Geocoding-related Parameters

ProximityRadiusSearchDefaultMaxResultCount

Maximum number of results to return if performing a radius (n miles or kilometers) search from ClaimCenter. This parameter has no effect on ordinal (nearest n) proximity searches. This parameter does not have to match the value of the corresponding parameter in the `ContactManager config.xml` file.

Default: 1000

ProximitySearchOrdinalMaxDistance

The maximum allowable distance to use if performing a proximity search for the *nearest-n* contacts or users. (This is also known as an *ordinal* search.) By default, ClaimCenter treats the allowable distance value as miles. However, if the system default unit of distance (as controlled by `UseMetricDistancesByDefault`) is set to `true`,

ClaimCenter treats this values as kilometers. The `ProximitySearchOrdinalMaxDistance` parameter **only** affects ordinal proximity (nearest-n) searches, meaning searches that are **not** based on distance or on searches that use a radius.

IMPORTANT This parameter has no effect on radius-based searches (*n*-miles) or walking-the-group-tree-based proximity assignment.

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.

Default: 300

UseGeocodingInAddressBook

If `true`, ClaimCenter enables geographical data and proximity search on the [Address Book](#) page. See `UseGeocodingInPrimaryApp` to enable searches from other pages.

Default: False

UseGeocodingInPrimaryApp

If `true`, ClaimCenter enables geographical data and proximity search on application pages. See `UseGeocodingInAddressBook` to enable searches from ContactCenter.

Default: False

UseMetricDistancesByDefault

By default, ClaimCenter uses miles and United States distances for driving distance or directions. If set to `true`, the application uses kilometers or metric distances instead.

Default: False

Integration Parameters

ContactAutoSyncWorkItemChunkSize

If you integrate Guidewire ClaimCenter with Guidewire ContactManager, then it is necessary to maintain the synchronization of contacts between the two Guidewire applications. Batch process `ContactAutoSync` controls this synchronization. (See “[ContactAutoSync](#)” on page 138 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

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 at the time of receiving the notification.

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

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

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

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

- 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.

Miscellaneous Bulk Invoice Activity Pattern Parameters

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. (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. (It was not possible to update the status from Pending-void or Voided to Issued or Cleared).

Required: Yes

Miscellaneous Financial Activity Parameters

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

AllowBulkInvoiceItemsToHaveNegativeAmounts

Within ClaimCenter, it is possible to make negative and zero dollar transactions and checks, which you can use to make negative payments. This is possible, however, for individual transactions only. By setting parameter `AllowBulkInvoiceItemsToHaveNegativeAmounts` to `true`, ClaimCenter permits bulk invoice items to have negative values and disables the bulk invoice validation checks associated with negative values.

IMPORTANT If you set this value to `true`, then Guidewire **strongly** recommends that you configure the PCF and business rules to enforce valid transaction values. With this parameter set to `true`, there is an increased risk of invalid bulk invoices.

Default: False

AllowSoapWebServiceNamespaceCollisions

Within Guidewire ClaimCenter, it is invalid to publish two web service types with the same name in the set of types for method arguments and return types. This is true even if the types have different packages and even if they are in different published web services.

ClaimCenter provides the `IgnoreSoapWebServiceReferenceNamespaceCollisions` configuration parameter to make development due to this problem easier.

- If set to `false` (the default), any name space collision generates an error message and ClaimCenter blocks application server start-up.
- If set to `true`, these error messages become warnings instead. *Use this setting only for development and debugging until you have time to rename your classes to fix the namespace collision.*

See “Web Service Types Must Have Unique Names” on page 31 in the *Integration Guide* for more information.

WARNING It is unsafe to set `AllowSoapWebServiceReferenceNamespaceCollisions` to `true` for production servers. If you have an special case in which it is difficult to rename classes to avoid name-space collisions, please contact Guidewire Customer Support.

Default: False

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. See “Claim Number Generator Plugin” on page 333 in the *Integration Guide* for more information.

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

LegacyExternalEntityArraySupport

Set to `true` to expose external entity collection types as arrays. Use this to support the existing Java plugin implementations without any code upgrade.

Default: False

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

StyleReportURL

The URL for the InetSoft StyleReport server. A `null` value is acceptable.

Default: *None*

Set for Environment: Yes

PDF Print Settings Parameters

See also “`PDFMergeHandlerLicenseKey`” on page 47.

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/>

For a discussion of FOP, see also “Printing in Non-US Character Sets” on page 480 in the *Configuration Guide*.

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

Scheduler and Workflow Parameters

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 129 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

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

DisableCBQTForClaimSearch

ClaimCenter works around some bad execution plans by disabling optimizer cost base transformation 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 effects only Oracle databases.

Default: True

DisableCBQTForTeamGroupActivities

ClaimCenter works around optimizer cost base transformation 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

DisableHashJoinForClaimSearch

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

Default: True

DisableHashJoinForProximitySearch

This parameter enables you to disable the use of hash joins while performing a proximity search using the Oracle database. The default value for the parameters is `false`. This parameter has no effect on databases other than Oracle. This parameter has no effect on databases other than Oracle. This parameter has no effect if you are not using proximity search.

Default: `False`

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 effects only the Oracle database.

Default: `True`

DisableIndexFastFullScanForProximitySearch

This parameter enables you to disable the index fast full scan while performing a proximity search using the Oracle database. The default value for the parameters is `false`. This parameter has no effect on databases other than Oracle. This parameter has no effect if you are not using proximity search.

Default: `False`

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 effects only the Oracle database.

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

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: 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

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 (thousands), the extra join can be better. *This parameter chooses at what point the query code switches from using direct parameters to using extra joins.*

It can take the following values:

- A value of -1 (or any negative value) 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 positive value 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.

SQL Server. For those ClaimCenter installations that use SQL Server as the database, Guidewire recommends the following:

- 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

ShouldSynchUserRolesInLDAP

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

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

Spellcheck Parameters

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

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 this to 10 to calculate statistics for the last 10 days, including today. You can also set this 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

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

HidePolicyObjectsWithNoCovrancesForLossTypes

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 coverances 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.

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)

ListPageSizeDefault

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.

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: Ocean

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

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 a work item before marking it as *failed*.

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

part II

The Guidewire Development Environment

Working with Guidewire Studio

This topic describes Guidewire Studio and the Studio development environment.

This topic includes:

- “What Is Guidewire Studio?” on page 83
- “Starting Guidewire Studio” on page 84
- “The Studio Development Environment” on page 85
- “Working with the QuickStart Development Server” on page 86
- “ClaimCenter Configuration Files” on page 88
- “ClaimCenter Resources Tree” on page 90
- “Configuring Diagnostic Logging in Studio” on page 90
- “Configuring Guidewire Studio” on page 91
- “Linking Studio to a SCM System” on page 91
- “Setting Font Display Options” on page 96
- “Setting Code Completion Options” on page 97
- “Setting Server Default Options” on page 99
- “Configuring External Editors” on page 101
- “Setting the Studio Locale” on page 103

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

Guidewire supports running Studio on the Microsoft Windows operating system only. To install Guidewire Studio, follow the installation directions in the *ClaimCenter Installation Guide*.

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 Microsoft Windows command line.

IMPORTANT Guidewire supports Guidewire Studio running on the Microsoft Windows platform only.

To start Guidewire Studio

1. Verify that your JAVA_HOME environment variable is set to the correct Java home directory. (See “Installation Environments Overview” on page 12 in the *Installation Guide* for details.)
2. 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 starts exceedingly slowly as it must load a large amount of type information. Subsequent starts, however, generally load much more quickly.

Running Studio in 64-bit Mode

In the base configuration, Studio starts in 32-bit mode. To run Studio in 64-bit mode, do the following:

1. Locate and open `studio.bat` for editing, then add the following line:

```
set JAVA_HOME=64-bit JDK directory
```

Set the value of JAVA_HOME to the root of your 64-bit JDK directory.
2. Locate and open `studio.gs` for editing, then add the following line within protected function `runImpl`:

```
prog.addJvmArg("-XX:+UseCompressedOops")
```
3. Start Studio as you would otherwise.

To stop Guidewire Studio

To stop Guidewire Studio, simply 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). However, Guidewire recommend that you use the **File** menu instead.

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.

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

The Studio Development Environment

Note: See “Installation Environments Overview” on page 12 in the *Installation Guide* for information on the software and hardware requirements for ClaimCenter Studio.

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.)

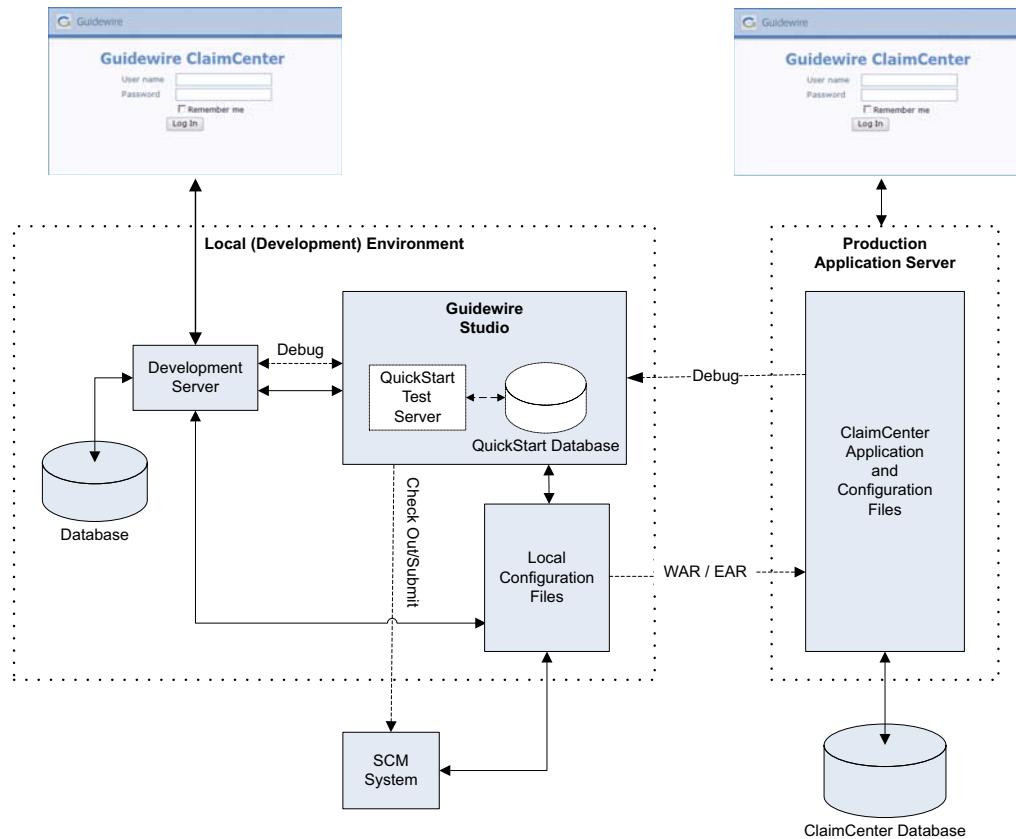
IMPORTANT Guidewire recommends that you do **not** run Studio on a machine with an encrypted hard drive. If you run Guidewire Studio on a machine with hard drive encryption, Studio can take up to 15+ seconds to refresh. This can happen as you switch focus from the Studio window to something else (such 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

Note: If desired, 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 56 in the *Installation Guide*.

You cannot start the QuickStart development server directly from Studio. (You cannot manually start the QuickStart test server as 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 69 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
ContactCenter	8280
BillingCenter	8580

Note: For more information on the gwcc dev commands, see “Installing the QuickStart Development Environment” on page 32 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 under the following circumstances:

- If things get odd from hot-deploying scripting resources (rules or other resources).
- 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

IMPORTANT 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.

Note: You do not actually need to be connected to the server from Studio to reload PCF files, although it does not hurt.

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
admin	Contains administrative tools. See “ClaimCenter Administrative Commands” on page 169 for descriptions.
bin	Contains the gwcc batch file and shell script used to launch commands for building and deploying. See “Commands Reference” on page 69 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.
java-api	Contains the Java API libraries created by running the gwcc regen-java-api command. See “Regenerating the Integration Libraries” on page 17 in the <i>Integration Guide</i> .
modules	Contains subdirectories including configuration resources for each application component.
soap-api	Contains the SOAP API libraries created by running the gwcc regen-soap-api command. See “Regenerating the Integration Libraries” on page 17 in the <i>Integration Guide</i> .
studio	Contains Studio preferences and TypeInfo database caches. Studio generates this directory when you first launch Studio.
webapps	Contains necessary files for using QuickStart or Tomcat application servers or WebLogic and WebSphere application servers for development.

How ClaimCenter Interprets Modules

ClaimCenter groups configuration resources in module folders within the `ClaimCenter/modules` directory. ClaimCenter evaluates these resources at startup according to a specific order. ClaimCenter always checks first for a resource in the `configuration` module.

The module directories can contain distinct copies of the same resource file. In that case, the predominant copy is the first one ClaimCenter finds. ClaimCenter disregards any others.

For example, in the base install, the file `Desktop.pcf` resides in the following directory:

```
ClaimCenter/modules/cc/config/web/pcf/desktop
```

If you edit this file from Guidewire Studio, Studio places a new copy in the `configuration` directory. The original file remains, but ClaimCenter ignores it for as long as the edited one exists. If you delete the edited copy from the `configuration` module, ClaimCenter uses the copy in the `cc` module.

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.

If you use Guidewire Studio to edit a configuration file, Studio automatically copies the file to the `configuration` module, if a copy is not already present. 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`.

For instructions on building and deploying ClaimCenter, see the *ClaimCenter Installation Guide*.

Studio File Structure

Studio presents a virtual directory structure in its `Resources` pane. It flattens all the configuration files into a single `configuration` module. Studio transparently manages these files and their (editable) copies. You need not be concerned about which directory a file resides, Studio manages this for you. If you add a file (for example, a new email or note template), only place it in the appropriate place in the `modules/configuration` directory. Studio monitors this directory and displays all new files in the `Resources` pane as you add them.

Recovering from Incorrect Data Model Changes

If you incorrectly modify a data model configuration file, Studio refuses to open the next time you attempt to start it. This is because it cannot build the type system without a valid data model. Instead, it provides an `Exception`.

tions dialog that provides information about the error, including the name of the problem file and the text of the error message. You must correct the error before Studio can start correctly.

The **Exceptions** dialog also contains a **Details** button that (if clicked) opens another dialog that provides more information about the problem. It also contains a button that (if clicked) opens the problem file in an external editor for ease in correcting the problem. (You must have linked an external editor to the file type in Studio for this to work. To link an external editor to a file type, see “Configuring External Editors” on page 101.)

Entity type system corruption. Occasionally, the entity type system itself becomes corrupt. This condition can cause Studio to hang (to refuse to complete the initialization process). You can correct this condition by deleting the following ClaimCenter application folder and all its contents:

ClaimCenter/Studio/db

Deleting this folder forces Studio to rebuild the type system the next time that you open Studio, which recreates the files in this folder.

ClaimCenter Resources Tree

The left-hand pane in the Studio window displays a list of ClaimCenter resources that you can access through ClaimCenter Studio. This is the **Resources** tree, which groups the various ClaimCenter resources into folders or nodes. Clicking a node expands that node. In many cases, it opens an editor for that resource type as well.

Directly above the **Resources** tree, you see a drop-down for filtering the visible resources. The drop-down provides you with the following choices:

Filter	Displays
All resources	(Default) All resources, both active and inactive.
Edited here	Only those resources that you newly created in the configuration module or those that you copied-on-edit to the configuration module. This filter is available in configuration or development mode only.
Edited now	Only those resources that are currently open for edit, according to the active version control system.

See Also

- “Working with Studio Resources” on page 120

Configuring Diagnostic Logging in Studio

Guidewire Studio provides the ability to enable additional diagnostic logging in the Studio console. In the base configuration, this includes timing information on certain specific actions in Studio only. For example, the following log entries indicate that focus switched between applications to activate the Studio main window. The last entry indicates that a different view tab became active in Studio.

```
[java] 2010-09-08 09:02:29,234 DEBUG Deactivate Task: 256ms
[java] 2010-09-08 09:02:37,711 DEBUG Activation Task: 264ms
[java] 2010-09-08 09:02:37,819 DEBUG selectTab(...) : 12ms
...
```

In the base configuration, Guidewire disables this functionality by design. In general, there is no need to enable this functionality unless Guidewire instructs you to do so.

Note: You can find information on application logging in “Configuring Logging” on page 35 in the *System Administration Guide*.

To enable diagnostic logging in Studio

1. Start Guidewire Studio, if it is not already running.

2. Within the **Resources** tree, navigate to **Other Resource** → **logging**.
3. Select **New** → **Other file** from the contextual right-click menu.
4. Enter **studio.properties** in the **New File** dialog. Studio creates a blank file with this name and places it in the **logging** folder. You see this file in the right-hand side of the screen, in a view tab labeled **studio.properties**.
5. Enter the following in this file.

```
log4j.category.Studio=DEBUG
```

This text is case-insensitive.
6. Save your work and restart Studio. Studio now logs all actions in a time-stamped list in the Studio console window.

Notes

- Studio enables diagnostic logging only if you enter the required text exactly as indicated.
- Studio disables this functionality if you remove all text from the file or if you deviate from the standard text (other than case).

Configuring Guidewire Studio

There are a number of configuration tasks that you can access from the Studio **Tools** → **Options** menu. These include:

- Linking Studio to a SCM System
- Setting Code Completion Options
- Configuring External Editors
- Setting Font Display Options
- Setting Server Default Options
- Setting the Studio Locale

Linking Studio to a SCM System

Guidewire recommends that you maintain your ClaimCenter configuration files in a Software Configuration Management (SCM) system such as Perforce. Using such a system ensures that you can save, manage, and restore any modifications that you make to a file. Guidewire provides support for the Perforce, CVS, and SVN (Subversion) version control systems. Guidewire also provides support for a generic (non-specific) version control system.

If you have such a system, Guidewire strongly recommends that you add your ClaimCenter installation directory and files to it as part of setting up your configuration environment. If you store the Studio configuration files in a SCM system, Studio can do the following for you:

- Mark a newly created file to add to your version control system on the next file checkin.
- Check out the file for editing if you modify that file from within Studio.
- Revert your file to the last checked in version to reverse your changes. (This is not true for all SCM systems.)
- Synchronize the file to the latest version in your SCM system. (This is not true for all SCM systems.)
- Submit your version of a file to the SCM system.

As with any set of files under version control, there are cases in which two different people edit a file concurrently, thus resulting in a conflict during file check-in. It is the responsibility of the authors to resolve these conflicts before checking in changes.

Linking Studio to a Software Configuration Management (SCM) System

Linking Studio to a SCM is a two-step process:

1. You must first set up the SCM-Studio association in Studio itself. See “Configuring Version Control” on page 94 for details on linking Studio to one of the supported SCM systems.
2. You must then modify `ClaimCenter/modules/ant/build.properties` to include a `scm.regex` statement. There are several (commented-out) examples of this statement in the file, one for CVS and one for SVN (Subversion). Studio uses the `scm.regex` statement in determining which file directories to exclude in calculating the checksum for source-controlled files.

Studio and Your Version Control System

Check the application product files into a Guidewire-supported SCM system as specified in its documentation. Check all Guidewire application files into your SCM system. However, the only files you ever need modify and need to check in are files in the `module/configuration/config/` directory. As you open and edit files in Studio, Studio places copies of the files in this directory. It is these files that Studio checks into your version control system, if you select that option in the Studio menu.

WARNING If you modify a base application file in any directory other than `modules/configuration`, you can damage ClaimCenter, so much so that it can refuse to start.

IMPORTANT Guidewire **strongly** recommends that you set *all* of the ClaimCenter application directories and files to read-only *except* for the `modules/configuration` directory and its files. (This is especially true if you are not using a SCM to manage your files.) This ensures that you do not inadvertently change one of the application base files, which can invalidate your ClaimCenter install.

Version Control Systems and Keyword Substitution

Certain SCM systems perform keyword substitution on text strings that they encounter in a file (for example, replacing `##DATE` with the current date).

- If you use a SCM system that performs keyword substitution, you must turn off this feature for your initial check-in of the ClaimCenter application files. Otherwise, it can invalidate the configuration checksum, which invalidates your ClaimCenter install.
- If it is not possible to turn off this feature, then zip all the application files and check the zipped file into your SCM system.

Accessing the Source Control Commands

To access the source control commands, select a file in the Resources tree and select **Source Control** from the **File** right-click menu. You see the following commands:

Command	Use to
Add	Mark a newly created file to “add” to the source control system.
Edit	Open (check out) a file for editing. If the file is a base configuration file, Studio makes a copy of the file in the <code>modules/configuration</code> directory. It is this file that you edit, and, it is this file that you must check into source control.
Revert	Revert a file back to its previously checked-in version in source control.
Sync	Synchronize your local copy of the file with the last previously checked-in version of the file.
Submit	Submit your local (modified) version of a file to the SCM system.

Note: Guidewire supports the **Add**, **Edit**, **Revert**, **Sync**, and **Submit** commands for the CVS and SVN source control management systems. Studio supports all of these commands for P4 (Perforce) *except* for **Submit**.

Submitting Files to the SCM System

If you use a SCM system to manage your files, ClaimCenter gives you option of submitting changed files to the SCM system using a Studio menu command. The **Submit** menu command is available only if you have opened the resource for editing or adding to source control. Selecting the **Submit** command opens the **Submit** dialog. Within the **Submit** dialog, Studio selects your resource by default for submission to the SCM system. From within this same dialog, you also have the option of submitting “All files”, which submits all files currently open for edit or add to the SCM system.

Deleting Files

On the **Edit** menu, you see a **Delete** or a **Revert to Base** command after selecting a resource file, depending on the context.

- If you created the file, then you see the **Delete** command, and you can delete your local copy of the file.
- If the file is part of the base configuration, then Studio does not permit you to delete the file. If you have made a copy of the file, you can see **Revert to Base**. If used, Studio deletes your modified copy of the file, and, thereafter, uses the base configuration version of the file.
- If a file exists only in the base configuration, then Studio disables these commands. (You can neither delete the base configuration file nor delete a local copy of the file.)

Viewing the SCM Log

You can view a copy of the SCM log by navigating to the Studio **Tools** menu, then selecting **SCM Console**. Studio opens a popup window that displays the SCM commands and error messages as they occur.

IMPORTANT Guidewire recommends that you do the following the first time that you try to use Studio with a CVS, SVN, or P4 software configuration management system. Before continuing, open up the SCM Console and verify that the SCM system is working properly. The SCM Console displays all commands issued by Studio to the SCM system, and any error messages that the SVN server might throw. From the SCM Console, you can determine if your SCM adapter is working correctly or not.

To edit a SCM file within Studio

1. Select a resource in the **Resources** tree. If the file is not already checked out, Studio opens a dialog box that informs you the file is not open for edit. It then asks you whether you want to edit that file. (Studio highlights the names of checked-out resource files in blue.)
2. Select **Yes** if you want to continue editing or **No** otherwise.
 - If you select **Yes**, Studio automatically checks out the file from the SCM system so that you can edit it.
 - If you select **No**, Studio opens a non-editable, read-only version of the file.If you have not edited the file before, Studio creates a copy of the (read-only) base configuration file in the appropriate `/modules/configuration/config` directory. This is the file that you edit.
3. Submit the modified file directly to the SCM system to save the file after you finish editing it.

You can also mark a file for deletion or revert it to a previous version. You do this by selecting the file in the Studio **Resources** pane, selecting **Source Control** from the right-click menu, and then the appropriate command.

Note: Studio colors the file names in the **Resources** tree various colors to indicate its state, for example, red, green, blue, purple, and black. For a discussion of what these colors mean, see “Working with Studio Resources” on page 120.

Configuring Version Control

To set how Studio interacts with a SCM system, navigate to **Tools** → **Options** to open the **Configuration Settings** dialog box, then select **Version Control** from **Project Settings**. Use this dialog to select the SCM system and to set any necessary connection parameters.

In the base configuration, Guidewire provides the following version control modules that you can use to integrate Studio with a supported SCM system:

- CVS Module
- File Filter Adapter Module
- File System Module
- Perforce Module
- Subversion

You need to set connection parameters for most of these.

Note: Studio tests the connection to your SCM system as you exit the **Version Control** dialog. If it cannot make a connection, Studio generates an error. You must correct the error before you can exit the dialog.

After you integrate Studio with your SCM system, you can view the output of the software configuration management system in Studio **SCM Console** (which is also under the **Tools** menu).

CVS Module

Note: If you select CVS as the SCM system, Studio disables the **Edit** option in the **Source Control** right-click menu. This is because CVS does not have the concept of opening a file for edit. However, if a file is writable, Studio considers it to be in the *open for edit* state and paints the resource blue in the resource tree. Also, if you check a file into a CVS source control management system, then you cannot revert that file through Studio.

If using CVS (Concurrent Versions System) as your SCM system, then do the following *before* you launch Studio and make your selection in the **Version Control** dialog:

- Log into the CVS server at least one time and check out files from the CVS server before you open Studio. This negates the need to ask for the username, password and server connection parameters from within Studio. This does not mean that the files must remain checked out at the time you run Studio. It simply means that you must have previously checked them out, followed by checking them back in or reverting them.
- Set the CVSROOT environment variable before launching Studio. The CVSROOT environment variable contains connection information to the CVS server, such as `:pserver:username@host/some/dir`. (The CVS server requires this value to check out files for the first time.)
- Verify that you have the command line `cvs.exe` client software installed.

After you select the CVS option in Studio, enter the full path to the CVS executable and the client root. The *client root* is the local path of root directory `ClaimCenter/modules/configuration`. (This is the only application directory that you ever need modify.)

Guidewire recommends that you check out your CVS repositories as read-only. To enforce this, use the `-r` global option during checkout, for example:

```
cvs -r checkout
```

File Filter Adapter Module

IMPORTANT If you use a non-Guidewire-supported version control system, then Guidewire recommends that you use this adapter. Do not attempt to write your own SCM adapter. Guidewire specifically does *not* support any attempt to create your own SCM adapter.

If you use a SCM system for which Guidewire does not provide a specified version control module, then select the **File Filter Adapter** module. This allows you to skip the control files specific to your system with a regular expression. Studio ignores all files whose absolute name matches the given regular expression.

Although not recommended, you can also use this SCM adapter for both the Subversion (SVN) and CVS source control systems. To do so, you need to provide a regular expression to indicate which files and folders to ignore.

- If using a CVS system, enter the following as the regular expression:
.*CVS.*
- If using SVN, enter a regular expression similar to the following:
.*/*.svn(\\.*)*

File System Module

If you do not select a SCM system, Studio defaults the **Version Control Module** to **File System**. If you set a file or folder within the file system as read-only, Studio removes the read-only flag on the file if you select **Source Control → Edit**.

As with the other options, Studio indicates files or folders that you open for editing with a blue color in the **Resources** tree.

Perforce Module

Note: It is possible that you also need to set a P4CLIENT environment variable for the Microsoft Windows' operating system. To do so, open the **System Properties → Environment Variables** dialog, find (or create) the **P4CLIENT** variable, and enter the name of your local machine. You must perform this action and also configure Guidewire Studio to recognize the Perforce client.

If you select the **Perforce** module, Studio presents you with several different options. You can either choose to use the default Perforce configuration, or to set specific Perforce configuration parameters yourself. In either case, you must set the path to the P4 executable.

Use system's default Perforce configuration. If you select this option, you need only enter the path to the P4 executable. Studio uses the system's default Perforce configuration.

Specify different Perforce settings. If you select this option, then you must also enter values for the following fields.

- **Port** - The default is <p4 server>:1666.
- **Client** - The name of the Perforce ClientSpec.
- **User and Password** - Valid username and password to use to log into Perforce.

Path to P4 executable. Enter the path to your Perforce executable:

- If you have a **PERFORCE_HOME** environment variable set, then you can accept the default p4.
- If you do not have a **PERFORCE_HOME** environment variable set, enter the full path. For example: C:\Program Files\Perforce.

Note: Guidewire supports the **Add**, **Edit**, **Revert**, and **Synch** operations for P4 (Perforce). Studio does *not* support the **Submit** operation with Perforce.

Subversion

The Studio Subversion (SVN) version control module is very similar in operation to the CVS version control module. It makes many of the same assumptions.

Like CVS:

- You must check out all your files from source control at least one time before using Studio. See "CVS Module" on page 94 for details.

- You must also choose the (default) option to keep the user's credentials locally. Studio performs no authentication with the remote server and expects the command line executable to do it.
- The **Sync** operation is not available in Studio.
- A file is *open for edit* if the file is writable. The **Edit** operation is not available in Studio.
- You need to enter the client root path, and the path to the SCM executable.

Unlike CVS, however, you can revert a file.

To configure Studio to work with the Subversion SCM system, enter the path to the Subversion executable and the path to the client root. The *client root* is the local path of root directory *ClaimCenter/modules/configuration*. (This is the only application directory that you ever need to modify.)

You must enter the full path to the configuration module. For example, you would enter something similar to the following:

C:\Guidewire\workspace\cc\modules\configuration

You must also enter the full path to the SVN executable. For example, you might enter something similar to the following:

C:\Program Files\CollabNet Subversion\svn.exe

IMPORTANT Guidewire *only* supports Subversion clients that provide a command line interface. Studio needs this hook to implement the source control functionality. If your Subversion client does not support a command line interface, then you cannot use it with Guidewire Studio. For example, you cannot use the open source *TortoiseSVN* as a Subversion client as it does not support a command line interface.

Setting File Update and Deletion Parameters (General Settings Tab)

Use the fields on this tab to specify whether Studio also updates the SCM change list as you add or delete a resource. For example, if you add a new rule, Studio creates a corresponding file in the *modules/configuration/config/resources* directory. If you instruct it, Studio can also modify the SCM change list and indicate that the file needs adding to source control (*Open for add*).

The following list describes the options that you can set on this tab.

Option	Description
Show options before adding or deleting	Open the SCM file add (delete) options dialog before modifying the SCM change list.
Add or delete silently	Add the file to the SCM change list for addition (or deletion) without further human action.
Do not add or delete	Do not add the file to the SCM change list.
Show "Clear read-only Status" Dialog	Display a dialog for read-only files to enable you to clear the read-only flag.

Setting Font Display Options

To set how Studio handles various font and Gosu editor options, do the following. First, navigate to **Tools** → **Options** to open the **Configuration Settings** dialog box, then select **Colors & Fonts** from **Project Settings**. Use this dialog to set the how Studio displays text in the Gosu editor. You can set the font type and size of the Gosu code, for example. You can also set how Studio displays specific Gosu code items, such as keywords or operators. Studio displays a code sample at the bottom of the dialog that reflects your settings so that you can view the effect of your choices immediately.

Using this dialog, you can set the following:

- Text font, size, and anti-alias properties

- Character format properties
- Foreground and background colors of various Gosu elements

You can also enable the use of the mouse wheel to increase or decrease the font size.

Setting Code Completion Options

To set how Studio handles various coding options, navigate to **Tools** → **Options** to open the **Configuration Settings** dialog box, then select **Code Completion** from **Project Settings**. Use this dialog to set how Studio handles deprecated methods and the transformation of Java code into Gosu:

- Allow Deprecated References
- SmartFix

Allow Deprecated References

Studio provides the means for you to enable or disable viewing of deprecated items. Deprecation usually refers to the gradual phasing-out of a software or programming language feature. Over time, a method or property can become obsolete and Guidewire discourages its use. However, it may still be possible to use the deprecated item so as to provide backward compatibility. If in doubt, consult the Studio API reference to determine if a method or property is still valid.

You can choose to either view or hide deprecated items in the Studio **SmartHelp** and the **Complete Code** windows. Choosing **Tools** → **Options**, then **Code Completion**, opens the following dialog box:

In the dialog box, select **Allow deprecated references** to display deprecated items within Studio. If made visible, Studio displays a deprecated item in **SmartHelp** by placing a line through it. It is still possible to select the deprecated item, however.

Studio indicates a deprecated item using one of the following methods:

- In **SmartHelp**, Studio strikes a line through the deprecated item.
- In Gosu code (within an editor), Studio underlines the deprecated item.

Studio also turns the validation status indicator to yellow if the Gosu code is valid but contains warnings. However, if you insert a deprecated item into a rule, it is always visible in the rule, regardless of the status of this setting.

SmartFix

Use the Studio **SmartFix** settings to set the level of automatic code completion with the Studio editors. Checking an option activates it. Guidewire provides the following **SmartFix** settings:

- Add import for unrecognized symbol
- Make implicit coercion explicit
- Fix Java-style type cast
- Create unrecognized display key
- Convert string literal to new display key
- Change old constructor syntax to new
- Remove unused variable
- Add missing 'override' modifier
- Implement functions and properties

Add import for unrecognized symbol

This SmartFix adds a Gosu `uses` statement to import the type if you enter a name that is a valid relative type name. For example, if you type:

```
var x : StringBuilder
```

SmartFix adds the following `uses` statement to your Gosu class:

```
uses java.util.StringBuilder
```

Make implicit coercion explicit

This SmartFix detects coercion warnings, and, if invoked, appropriately casts the implicitly coerced expression to make it explicit. For example, if you enter:

```
var x : String  
var y = 42  
x = y // coercion warning
```

SmartFix casts the last statement to make it explicit:

```
x = y as String
```

Fix Java-style type cast

This SmartFix detects Java-style casting and changes it to Gosu style casting. For example, if you enter:

```
(String)42
```

SmartFix converts this to Gosu casting

```
42 as String
```

Create unrecognized display key

This SmartFix detects a compile error on an unrecognized display key and automatically creates one. For example, if you enter:

```
displaykey.Example // Example does not exist yet
```

SmartFix automatically creates the display key for you and lets you edit the name.

Convert string literal to new display key

This SmartFix detects string literals and prompts you to create a display key for it. This is to aid in making Gosu code more localizable.

Change old constructor syntax to new

This SmartFix detects old-style constructor syntax and automatically converts it to the new style. For example, if you enter:

```
class MyGosuClass {  
    MyGosuClass ()  
    ...  
}
```

SmartFix automatically makes the following conversion:

```
class MyGosuClass {  
    construct()  
    ...  
}
```

Remove unused variable

This SmartFix detects unused variables and automatically removes them.

Add missing 'override' modifier

This SmartFix detects a missing `override` modifier on methods and properties that override a `super` member. For example, if you enter:

```
function OverrideThisFunction()
```

SmartFix automatically changes this to the following:

```
override function OverrideThisFunction()
```

Implement functions and properties

This SmartFix detects compile errors indicating a class has not implemented one or more methods and/or properties from an abstract super class or implemented interface. If invoked, Studio automatically generates default implementations for the missing methods (or properties) and inserts the missing items in the class.

Setting Server Default Options

Studio has two connection modes of operation:

- *Disconnected*, in which you can edit local resources through Studio, but there is no database connection. Because it lacks a database connection, you cannot use the Studio query-oriented features in disconnected mode. For example, code completion for activity pattern, user, or group values does not work in this mode. By default, Studio starts in *disconnected* mode.
- *Connected*, in which Studio interacts with the application server for debugging purposes. You almost never need to connect to a remote instance. If you do this mode, do so **only** for the purposes of debugging in Studio to help diagnose a problem. With that said, it is forbidden to connect Studio to a production server for the purposes of *tweaking* or configuring in any way. Guidewire specifically designs Studio to connect to a single local server.

Studio indicates its connection mode in the lower right-hand corner of the screen, along with the words **Disconnected** or **Connected**:

- **Disconnected** - 
- **Connected** - 

You can also determine the connection mode by opening the **File** menu and looking at the menu commands. There are two commands that pertain to the connection mode and that you can use to either establish a connection with the application (remote) server or to disconnect from it. The connection mode is the opposite of whichever one of these commands is active. (For example, if the **Connect to Server...** command is active, then Studio is operating in disconnected mode.)

Note: Studio does not automatically detect a running Guidewire application server, nor can it automatically connect to an application server. You must manually connect to the server.

Connecting to a running production server. *Guidewire requires that you provide valid user credentials (username and password) if you attempt to connect Studio to Guidewire ClaimCenter running in Production mode.* The user **must** have the **Administer Rules** permission (code, `ruleadmin`). Studio fails to connect if either the following are true:

- The user does not have valid credentials to authenticate.
- The user does not have the **Administer Rules** permission (code, `ruleadmin`).

You do **not** need to provide user credentials if the application server is running in Development or Test mode.

WARNING In general, Guidewire recommends that you **not attempt to connect Studio to a production server**. Use this option with extreme care and caution. This operation has the potential to modify production code and data. If you do so, Guidewire **strongly** recommends that you first set configuration parameter `ResourcesMutable` to `false`.

To connect to an application server

1. Open the login dialog using one of the following methods:

- Click **Disconnected** in the lower right-hand corner of the screen.
 - Select **Connect to Server...** from the Studio File menu.
2. Verify that your connection URL is correct. In the default configuration, this is some variation of the following:
- `http://localhost:8080/cc/`
3. (Optional) Select the check box next to **Log in as specific user** to connect as a specific user or user role. Selecting this option opens **User name** and **Password** fields. Enter a valid user name and password combination. This can be any valid combination, for any user role, not necessarily just supervisor or administrator roles. You must provide the valid user credentials to connect to a running application server in Production mode.

To set the default application server URL

1. Select **Options** from the Studio Tools menu. The **Configuration Settings** dialog opens.
 2. Select **Remote Server**.
 3. In **Connections Options**, enter a value for **Server URL**. Studio uses this value in the login dialog. Enter something similar to the following if the application server and the local file system physically exist on the same machine.

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

Note: You must enter the fully qualified domain if the application server and the local file system are not physically located on the same machine.
 4. (Optional) You can also enter a value in **Connect Script** that defines the file location of a script for Studio to use while connecting to a Guidewire application server:
 - Create the script as a shell command, for example: `run_my_server.bat`.
 - Include only commands directed at starting the Guidewire application server.
 - Place this script in the Studio working directory, which is `ClaimCenter/Studio`.
 - Enter the path in the **Connect Script** text box. For example, after you place file `run_my_server.bat` in the `temp` directory in the Studio folder, enter `/temp/run_my_server.bat` in the text box.
- Studio displays any errors it encounters in an error message dialog. It also echoes the error messages to the Studio console.
5. Click **OK** to save your work and close the **Configuration Settings** dialog.

Restarting the Application Server

If you modify some of the ClaimCenter resources in Guidewire Studio, you must restart the application server to force a database upgrade. You must also restart the application server if you delete, or revert to base, certain resources. Resources for which you must restart the application server include:

- Entities names
- Web services
- Quick Jump commands
- Plugins
- TypeLists
- Display keys
- Messaging environments
- Workflows

Studio indicates that you must restart the application server by placing a restart icon in the lower right-hand corner of the screen, next to the connection icon. 

In general:

- If you modify a Gosu rule, class, or enhancement, then you do *not* need to restart the application server. This is because Studio and the development application server share the same file space.
- If you modify a PCF file or a display key and want to see the changes within ClaimCenter, then log into ClaimCenter as an administrator. Then, select **Internal Tools** → **Reload PCF Files**.
- If you need to move data model changes to a production server, then you do the following:
 - Generate a .war (.ear) file.
 - Transfer it to the production server machine.
 - Restart the production application server.

Configuring External Editors

Studio contains built-in editors for many, if not most, of the files types that it manages. In general:

- For files extensions for which you have configured an external editor (through **Tools** → **Options**), Studio uses that file association to open an external editor.
- For file extensions that Studio recognizes, but for which it does not have a specific editor, Studio opens the file in its embedded text editor, in a view tab.
- For file extensions that Studio does not recognize *and* for which you have not configured an external editor, Studio passes the request to the Windows operating system. The Windows operating system then uses its default editor for that file type to open the file in an external editor.

File extensions that use the built-in text editor. Studio opens the following file types in its built-in text editor:

.css	.gst	.vm
.eix	.html	.xml
.eti	.java	.xsd
.etx	.js	.xsl
.gs	.properties	
.gsm	.txt	

File extension .gs is a special case:

- Within **Other Resources**, Studio opens .gs files in its embedded text editor.
- Within **Classes and Tests**, Studio opens .gs files within its built-in Gosu editor.

Associating an External Editor to a File Type

To instruct Studio to open an external editor to handle a specific file type, navigate to **Tools** → **Options** to open the **Configuration Settings** dialog box, then select **External Editors**. While you edit most Studio-managed resources directly within Studio, you must use an editor external to Studio to edit a small number of files.

You use the **External Editors** dialog to link a specific file type to a specific editor. To add an editor, click **Add** and enter the file extension and the path to the executable for your chosen editor.

At the minimum, Guidewire recommends that Studio users configure external editors for the following file types:

- *.gif
- *.jpg

- *.png

IMPORTANT If you use a third-party tool to edit ClaimCenter configuration files, Guidewire recommends that you work with one that fully supports Unicode character sets. If the tool does not handle Unicode characters correctly, it may 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 Unicode values correctly.

Working with an External XML Tool

It is possible to configure Studio to open .xml files directly in an XML editor that is external to Guidewire Studio. To facilitate this process, Guidewire provides an XML attribute named `xmllns`. The `xmllns` attribute defines the *namespace* for the XML elements in the file. This namespace (analogous to a package in Java) serves to further qualify a class name, primarily to avoid collisions between like-named elements from other schemas.

You can configure many XML editors to associate a namespace with an XSD. However, merely defining the namespace within Guidewire ClaimCenter is not sufficient to inform the XML editor which XSD to use in validating an XML document. You must configure the XML editor manually to associate the namespace with the XSD.

IMPORTANT The `xmllns` attribute is currently optional. However, Guidewire strongly recommends that you add it to your entity and typelist files as Guidewire reserves the right to make this attribute required in the future.

Entity files. Use the following for *entity* files (.eti and .etx):

```
<entity xmllns="http://guidewire.com/datamodel" ...
```

Typelist files. Use the following for *typelist* files (.tti and .txx):

```
<typelist xmllns="http://guidewire.com/typelists" ...
```

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. If the editing tool does not handle UTF-8 characters correctly, it can create errors. For XML files, it is possible to use a different file encoding, as long as you specify it in the XML prolog. For all other files, use UTF-8.

Configuring the External Difference Tool

You also use this dialog to set the tool to use in *diffing* (determining the differences) between two different versions of the same file. To set, browse to the executable for your difference tool. For example, if you use Araxis Merge as your difference tool, then you would browse to the following location:

```
C:\Program Files\Araxis\Araxis Merge v6.5\Merge.exe
```

To open the file in your chosen tool and view the differences, select a file in the Studio Resources pane, right-click, and then click **Source Control** → **Diff against base**.

WARNING Do **not** use a difference tool to modify ClaimCenter configuration files. This can potentially invalidate your ClaimCenter installation and make it unable to start.

Setting the Studio Locale

If you configure Studio for multiple locales, then you see an additional locale drop-down picker in the lower right-hand corner of the screen. Selecting a different locale changes the Studio labels (display keys) to reflect that locale *if you have translated versions of those display keys*. If you do not, then you see an error message for each display key for which Studio cannot find a translation.

See Also

- For information on configuring Studio for different locales, see “Localizing the Development Environment” on page 491.

ClaimCenter Studio and Gosu

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

This topic includes:

- “Gosu Building Blocks” on page 105
- “Gosu Case Sensitivity” on page 106
- “Working with Gosu in ClaimCenter Studio” on page 107
- “Gosu Packages” on page 107
- “Gosu Classes” on page 107
- “Gosu Enhancements” on page 110
- “The Guidewire XML Model” on page 111
- “Script Parameters” on page 111

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 169 in the *Gosu Reference Guide*.
- For information on the ClaimCenter base configuration classes see, “ClaimCenter Base Configuration Classes” on page 108.

- For information on the @export annotation and how it effects a class in Studio, see “Class Visibility in Studio” on page 109.
- For general information on Gosu enhancements, see “Enhancements” on page 209 in the *Gosu Reference Guide*.
- For information on using Gosu business rules within Guidewire ClaimCenter, see *ClaimCenter Rules Guide*.
- For information on script parameters and how to use them in Gosu code, see “Script Parameters” on page 111.

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:

- proper type names
- variable names
- keywords (such as var and if)
- method names
- property names
- package names
- all other language elements.

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 169 in the <i>Gosu Reference Guide</i>“Gosu Classes” on page 107
Interface	<ul style="list-style-type: none">“Interfaces” on page 191 in the <i>Gosu Reference Guide</i>
Enhancement	<ul style="list-style-type: none">“Enhancements” on page 209 in the <i>Gosu Reference Guide</i>“Gosu Enhancements” on page 110
Template	<ul style="list-style-type: none">“Gosu Templates” on page 291 in the <i>Gosu Reference Guide</i>
Package	<ul style="list-style-type: none">“Gosu Packages” on page 107
Guidewire XML Model	<ul style="list-style-type: none">“The Guidewire XML (GX) Modeler” on page 262 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`.

Note: For more information on Gosu classes, see the *Gosu Reference Guide*.

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 **Resources** tree.
3. Right-click, select **New**, then **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 108
- “Class Visibility in Studio” on page 109
- “Preloading Gosu Classes” on page 109

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:

- `gw`
- `libraries`
- `util`

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 *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 *libraries* Package

The `libraries` package contains a number of pre-built functions. To access these functions, either 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 also contains a number of pre-built classes that provide additional functionality.

The *gw.api.** Package

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 is visible in the Studio **Classes** folder in the **Resources** 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.

The `gw.plugin` Package

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

- See “Using the Plugins Editor” on page 141 for information on how to use Studio to work with plugins.
- See the *ClaimCenter Integration Guide* for an overview of the various types of plugins and how to implement a plugin.

Class Visibility in Studio

For a pre-built Gosu class to be directly visible Studio, Guidewire must mark that class with the `@export` annotation. It is possible that you can view a class file in the application file structure. However, without the `@export` annotation, you cannot directly view or access the class in the Studio Gosu editor.

In this case, 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:

```
ClaimCenter/modules/pc/gsrc/gw/webservice/pc/ccintegration/v2/ccentities
```

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	Static method invocations dictate some kind of action and have the following syntax : <code>type#method</code> The referenced method must be a static, no-argument method. However, the method can be on either a Java or Gosu type. 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: <code>gw.api.startup.PreloadActions#headerCompileAllGosuClasses</code> It is possible to add in your own static methods to use in this fashion as meets your business needs.
Type names	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: <ul style="list-style-type: none">Java - ClaimCenter loads the associated Java class file.Gosu - ClaimCenter parses the class completely.

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

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`.

As defined in the *Gosu Reference Guide*:

- 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.
- 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.

For more information, see the following:

- “[Classes](#)” on page 169 in the *Gosu Reference Guide*
- “[Enhancements](#)” on page 209 in the *Gosu Reference Guide*

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 `Resources` 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.

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 XSD to describe XML that conforms to your XML model. At run time, you can export XML for this type and—optionally—choose to export only data model fields that changed. If you have more than one integration point that uses a type, you can create different XML models for each type.

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

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

See Also

- For detailed information on the Guidewire XML model, see “The Guidewire XML (GX) Modeler” on page 262 in the *Gosu Reference Guide*. This section provides a complete example of how to use the Guidewire XML Modeler.

Script Parameters

Script parameters are Studio-managed resource that you can use as “global” variables within Gosu code, but which you manage through the ClaimCenter (not Studio) interface. ClaimCenter uses file `ScriptParameters.xml` as the system of record for script parameter definitions and default values. You can **only** create script parameters from within Studio (from **Tools → Script Parameters**). After creation, Studio adds the new script parameter to the `ScriptParameters.xml` configuration file.

On server startup, ClaimCenter compares the list of script parameters that currently reside in the database to those in the `ScriptParameters` file.

- ClaimCenter adds any script parameters that are in the XML file but not in the database to the database, with whatever initial values are set in the XML file.
- ClaimCenter ignores all other values in the XML file. This means that ClaimCenter explicitly *does not* propagate changes to values in the XML file to the database.

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

To repeat, Studio does not propagate to the database any changes that you make to script parameter values using the Studio **Tools → Script Parameters** dialog. *After you create a script parameter within Studio, you cannot use Studio to change its value. You must use the ClaimCenter Script Parameters administration screen.*

Script Parameters as Global Variables

There are several related reasons to create global variables:

1. You want a variable that is global in scope across the application, but one that you can actively administer (change or reset as needed) through the application interface.

2. You want a variable that can be used in any Gosu expression to hold a value. However, you also want the ability to change that value without having to edit the expression.

These two, while related, are entirely independent of each other.

- Use *script parameters* to create variables that you can administer through the ClaimCenter interface.
- Use *Gosu class variables* to create variables for use in Gosu expressions. (See the *Gosu Reference Guide* for information on Gosu class variables.)

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 value in Guidewire Studio.
- You administer script parameters and modify their values in the ClaimCenter interface (on the **Administration** tab).

IMPORTANT The application server references only the first (initial) value that you set in Guidewire Studio for a script parameter. Thereafter, it references the value set through the ClaimCenter interface and ignores subsequent changes to the value set through Studio.

To create a script parameter

1. In the **Tools** menu, click **Script Parameters**.
2. Click **New**.
3. In the **Name** box, type the parameter name.
4. In the **Type** list, select the parameter type.
5. In the **Value** box, set the parameter value. The value must be valid for the specified parameter type.
6. Click **Save**.

To delete a script parameter

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

1. In the **Tools** menu, click **Script Parameters**.
2. Select the parameter that you want to delete. (To find the parameter in the list, you can type part of the parameter name or click its type, and then click **Apply Filter**.)
3. Click **Delete**.

Referencing a Script Parameter in Gosu

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.

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

For example, the following Gosu 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
```

See also “Adding Comments” on page 38 in the *Rules Guide* for an example of using a script parameter to trigger rule logging.

Getting Started

This topic describes Guidewire Studio, which is the ClaimCenter administration tool for creating and managing ClaimCenter resources. (Studio resources include Gosu rules, classes, enhancements, script parameters, and the ClaimCenter data model files.) Using Guidewire Studio, you can do the following:

- 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
- Access rule sets, Gosu classes, and other resources stored in a SCM (Software Configuration Management) system

This topic includes:

- “Using the Studio Interface” on page 115
- “Working with Studio Resources” on page 120

IMPORTANT Guidewire supports Guidewire Studio running on the Microsoft Windows platform only.

Using the Studio Interface

The Guidewire Studio interface consists of either two or more panes arranged from left-to-right, with several of the right-most panes capable of being further subdivided.

Left-most (Resources) pane. The **Resources** pane lists the resources that you can manage through Guidewire Studio. This includes rules, classes, workflows, page configuration files, entity names, and display keys, for example. You can filter the resource list by selecting a filter from the drop-down list at the top of the **Resources** pane.

Center pane. The center pane of Studio changes depending on which resource you select in the **Resources** pane. For example, if you select **Page Configuration (PCF)**, you see the graphical PCF editor in the center pane.

Right-most pane. The right-hand pane of Studio changes depending on what editor is active in the center pane. For example, if you select **Page Configuration (PCF)**, you can select graphical elements to embed in the PCF file from the **Toolbox** in the right-hand pane.

Bottom pane. Depending on the editor involved, Studio can open a **Properties** pane at the bottom of the screen.

Resource tabs. Studio displays the most recently accessed resources as tabs just underneath the toolbar. To access a resource, select its tab. To close a tab, click the X at the right of the tab toolbar.

Validation status indicator. Studio indicates the validation status of Gosu code (in a rule or class, for example) to the right of the Gosu editor:

- Green indicates that the Gosu code is valid.
- Yellow indicates that the Gosu code is valid but contains warnings.
- Red indicates that the Gosu code is invalid. This code does not compile.

Using the Studio Menus

The Studio interface contains a set of menu items to aid you in managing Studio resources. Use these commands to create, edit, and debug the Gosu code used in the ClaimCenter business rules, Gosu classes, enhancements, and PCF files.

The following list describes the type of commands that you find in each of the Studio drop-down menu lists.

Menu	Contains commands to	See
File	Perform a variety of file-related and resource management functions: <ul style="list-style-type: none">• Create new Guidewire resources.• Connect to an external server• Import and export resources.	"Working in Guidewire Studio" on page 123
Edit	Perform standard text editing actions such as copy and paste.	"Using Text Editing Commands" on page 134
Search	Perform basic text find-and-replace actions.	"Using Find-and-Replace" on page 170
Code	Search for and navigate to different code entities.	"Using Studio Keyboard Shortcuts" on page 128
Rule	Create, delete, or rename a rule. This menu is only available if you select Rules in the Resources pane.	"Working in the Gosu Editor" on page 140
GUnit	Work with the GUnit tester.	"Using GUnit" on page 559
Debug	Start and stop the Studio debugging tool (the Gosu Tester), and set breakpoints and step through lines of code.	"Debugging and Testing Your Gosu Code" on page 551
Tools	Perform a variety of debugging-related and resource management functions: <ul style="list-style-type: none">• Open the Gosu Tester.• Manage Script Parameters.• Set Studio options (version control system, code completion options, linking external editors to certain file types, and similar options).	"Using the Gosu Tester" on page 555 "ClaimCenter Studio and Gosu" on page 105 "Configuring Guidewire Studio" on page 91
Window	Commands to manage the active Studio window, and display a list of the most recent Studio views.	"Managing Windows and Views" on page 118
Help	Commands to access the Guidewire Studio help features, including the Gosu API Reference.	"Entering Valid Code" on page 123

Using the Toolbar Icons

You can initiate many of the Studio menu commands by selecting the appropriate toolbar icon. You can roughly divide these icons into the following groups.

- Icons on the left-hand side activate commands that involve Guidewire resources.
- Icons on the right-hand side activate commands that involve the Studio debugging tool and Gosu tester.

The following list describes the toolbar icons that involve Guidewire resources. Many of these perform familiar text-processing functions and navigation operations.

Icon	Description	Shortcut
	Save your changes	Ctrl+S
	Delete the selection	Delete
	Undo the last operation	Ctrl+Z
	Redo the last undone operation	Ctrl+Y
	Copy the selection to the Studio clipboard and remove it from the interface	Ctrl+X
	Copy the selection to the Studio clipboard	Ctrl+C
	Paste the contents of the clipboard into Studio	Ctrl+V
	Copy the line of Gosu code at the present cursor location and past it directly underneath the source line.	Ctrl+D
	Undo last navigation operation	Alt+Left
	Redo last navigation operation	Alt+Right
	Find text or usage in the active view	Ctrl+F
	Verify (validate) a resource or resources	—

The following list describes the toolbar icons that involve the Studio GUnit tester.

- | | | |
|--|---|----------|
| | Activate the Edit Configuration command that opens the GUnit Run/Debug dialog. If you have GUnit tests defined, Studio presents a selectable drop-down list of your tests. | — |
| | Run the selected test configuration. | Ctrl+F10 |
| | Debug the selected test configuration. | Ctrl+F9 |

The following list describes the toolbar icons that involve the Studio debugging tool. For more information on the debugging tool and how to use it, see “Debugging and Testing Your Gosu Code” on page 551.

Icon	Description	Shortcut
	Stop debugging	Ctrl+F2
	Display menu of debuggers to start	—
	Display debuggers with which to intercept application execution	—
	Step to the next statement after the current execution point	F8

Icon	Description	Shortcut
	Step into the method call at the current execution point	F7
	Show the current execution point	Alt+F10
	Toggle a break point on the selected line	Ctrl+F8
	Toggle a break point on the selected rule	—
	Display the Gosu Tester window	—

Using the Right-Click Menu

Selecting an item in the **Resources** tree and right-clicking with the mouse opens a set of additional menus. These menu items are contextual, meaning that they change depending on the item selected. Some of the menu commands open additional submenus.

Menu command	Shortcut	Description
New		Opens a set of contextual submenus. You can create a new rule, class, or package, for example, depending on the item selected.
Delete	Delete key	Deletes the selected resource. Studio prompts you for confirmation. If this file exists in a SCM system, you may see the command Revert to Base , instead. See “Deleting Files” on page 93 for more information.
Rename Resource	Shift+F6	Renames a resource and all references to it throughout the set of Studio resources.
Move Resource	F6	Moves the resource to a different location in the Resources tree.
Find in Path...	Ctrl+Shift+F	Searches for a resource within Studio using the provided resource name or text string. For details of this command, see “Using Find-and-Replace” on page 170
Replace in Path...	Ctrl+Shift+R	Searches for a text string and replace its value with the supplied value. For details of this command, see “Using Find-and-Replace” on page 170
Verify Path		Verifies the validity of the Gosu code for the files in the supplied path. This opens a similar Verify dialog as from Tools → Verify . See “Validating Rules and Gosu Code” on page 173 for more information.
Diff Against Base		Runs a difference tool—if you have one defined—against this (modified) file and the equivalent base configuration file. You must have defined, or registered, an external difference tool in Studio for this command to work. See “Configuring External Editors” on page 101 for details on how to register an external difference tool with Studio (Araxis Merge, for example). IMPORTANT Do not use a difference tool to modify ClaimCenter configuration files. This can potentially invalidate your ClaimCenter installation and make it unable to start.
Source Control		Opens a set of additional submenus that perform software configuration management (SCM) commands, if you have linked Studio to a SCM system. For details, see “Linking Studio to a SCM System” on page 91.
Find in Explorer...		Opens up the Windows Explorer and navigates to the selected resource file.

Managing Windows and Views

Each time that you select an item from the left-hand **Resources** pane, Studio opens a view of the item in the right side of the screen. If you select more than one item (sequentially), Studio displays each item in its own separate tab. (Studio automatically saves a view as you switch to the next one.)

In the upper right-hand corner of the view area, there are several different icons that you use to manage your views. The following list describes them.

Icon	Description
	Opens a pop-up window that lists all currently open views. Selecting a view from the list brings that tab to the foreground.
	Minimizes the currently selected pane, maximizing all other panes.
	Maximizes all panes to the fullest extent possible.
	Closes—after automatically saving—the currently open view.

Guidewire Studio provides several different ways to navigate among its windows and views. To move between the various open views (tabs), do one of the following:

- On the **Window**, click **Back** (or **Forward**).
- Press ALT + (left/right arrow).
- Use an **Undo (Redo)** Navigation toolbar icon or .

To see a list of all the views (tabs) that you have opened recently, select **Recent Views** (CTRL + e) from the **Window** menu. This is useful if you have opened a large number of views and need to find a particular view among many. It is also useful if you closed a view and would like to open it again without searching through the resource tree.

Viewing multiple tabs. It is possible, as you work, that you may open many, many views (tabs). In this case, Studio may not be able to display all of the tabs properly due to viewing area constraints. If so, Studio displays the last partially visible tab with a tear-away look and an ellipsis (...).

If you hover your cursor over the ellipsis, Studio displays the entire tab row expanded for easy viewing. These tabs are fully functional. If you click one of these tabs, Studio moves you to that view.

Finding the active resource. Each open view contains a ClaimCenter resource (a class or a business rule, for example). You can find the resource in the **Resources** tree by using one of the following methods:

- ALT+F1 expands the **Resources** tree and highlights the file name of the currently active resource view.
- Right-clicking a view tab and selecting **Find in Resources** accomplishes the same action.
- Selecting **Find Selected View in Resources** from the toolbar **Window** menu accomplishes the same action.

Closing views. Studio provides several useful commands to manage the closing of one or more view tabs. Right-click a view tab and select one of the following from the menu:

- **Close This Tab**
- **Close All But This Tab**
- **Close All Tabs**

The *Window* Menu

The Studio toolbar contains a **Window** menu that contains a number of view-related commands. They are self-explanatory. The following table lists the **Window** menu commands.

Window menu commands	Shortcut
Forward	ALT+RIGHT ARROW
Back	ALT+LEFT ARROW
Next Tab	CTRL+TAB
Previous Tab	CTRL+SHIFT+TAB
Recent Views....	CTRL+E
Close Selected Tab	CTRL+F4
Close All Tabs	None
Close Other Tabs	None
Find Selected View in Resources	ALT+F1
Toggle Main Window	CTRL+SHIFT+F12

Note: Selecting a view tab and using the right-click menu provides you with an additional way to access several of the **Window** commands.

Working with Studio Resources

As discussed in “ClaimCenter Configuration Files” on page 88, Studio manages all ClaimCenter configuration and resource files. Do **not** attempt to modify Studio resource files manually. Instead, open the file from within Studio for editing. Studio thus manages the file within its file structure, within the *ClaimCenter/modules/configuration* directory.

WARNING Do **not** attempt to modify any files other than those in the */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.

The **Studio Resources** tree displays the ClaimCenter file structure as a flat or virtual file system. This means the physical location of a file does not matter. Studio groups the resource files into folders (or directories) in the **Resources** tree, but stores the files in multiple locations. Within the resources files, there are:

- Files that you can open and view, but not edit.
- Files that you can open, view, and edit.

Resource File Name Colors

Studio uses color to indicate the SCM status of a file. For example:

- The font color of a file name (node) in the resource tree corresponds to its SCM state.
- The color of a view tab corner corresponds to the SCM state of the corresponding resource, unless that resource is in the *not-open-for-edit* state.

The following list describes these colors and what they mean:

Color	Description
Black	The file has not been opened for editing. If a file is non-editable, its name always remains black.
Blue	The file has been opened for editing. The first time that you attempt to open an editable file, a Studio dialog asks if you want to open the file for editing.

Color	Description
Green	If you link the Studio file system to a SCM system, Studio automatically invokes the SCM Add function on any file that you create in Studio. A green file name indicates that you have created a file and that Studio has added it to the SCM system, but that the file has not yet been submitted. Studio then displays a Submit command on the (right-click) Source Control menu. <ul style="list-style-type: none"> • If you submit the file, Studio considers the file as checked into source control and displays the file name in black. • If you now open the file for editing from within Studio, Studio displays the file name in blue.
Red	If you link the Studio file system to a SCM system, Studio displays an Add command on the (right-click) Source Control menu. You use this command for files that you create outside of Studio, about which the SCM system knows nothing. <ul style="list-style-type: none"> • If you add the file to certain Studio directories, Studio recognizes the file and displays the file name in red. • If you invoke the Add command on this file, Studio invokes the SCM Add command on the file. It then changes the file name to green, indicating that the file has been added to the SCM system but not yet submitted. <p>IMPORTANT There are only very few directories for which Studio displays added files. One example is <i>Web Resources</i> → <i>web/templates</i>. In general, however, Guidewire does not recommend that you create files outside of Studio and manually add them to the Studio file system. Especially, do not add files to the <i>ClaimCenter/modules/cc</i> directory. Any attempt to add files to this directory can cause damage to the ClaimCenter application and prevent it from starting thereafter</p>
Purple	(Perforce) The file is editable on the local file system, but the SCM system is not aware that the file is open for edit. This applies to the Perforce SCM system only.

See Also

- “Linking Studio to a SCM System” on page 91

Resource File Tooltips

If you hold the mouse over a resource name in the **Resources** tree, Studio generates tooltips that provide additional information. This includes the file path and information related to the status of the file within any linked SCM system.

Validating Studio Resources

As you work with the various Studio resources, you can verify or validate your work. Studio provides specific menu commands that you can use to specify what you want to validate. Selecting **Verify** from the **Studio Tools** menu opens a dialog in which you can make the following verification choices. Unless explicitly stated otherwise, Studio automatically includes the resource children as well.

Option	Description
Verify changed resources	(Default) Performs validation on all resources that have changed since the last verify and all resources that had errors on the last verify operation. The list of resources that have changed also includes those resources that use other resources that have changed. Studio stores the fact that a type is erroneous in the TypeInfo database. Guidewire recommends in general that you use this option as it is much faster than a full verify.
Verify all resources	Performs validation on all Studio resources. This can be very slow.
Verify path:[resource name]	Performs validation on the selected resource and all of its children.
Verify view:[view name]	Performs validation on the resource represented by the currently selected view.

You can also access the **Verify** dialog by one of the following alternative means:

- By selecting the **Verify** icon from the toolbar. 
- By selecting a resource and choosing **Verify Path** from the right-click menu.

Studio displays any rule validation issues it encounters in a **Verification** pane that it opens at the bottom of the Studio interface. This pane presents the results of the verification process. You can click on a result item to navigate to that item.

The **Verification** pane contains the following:

- A **Show Errors** check box—unchecked this box removes nodes representing errors from the verification tree
- A **Show Warnings** check box—unchecked this box removes nodes representing warnings from the verification tree
- A filter text field—typing text in this field filters the nodes in the tree to only those that match the filter text and its descendants.
- A **Verify** icon—clicking this icon re-runs the verification with the same settings and update the results tree
- An **Expand All** icon—clicking this icon expands all nodes in the results tree
- A **Collapse All** icon—clicking this icon collapses all nodes in the results tree
- A **Close** icon—clicking this icon closes the current results pane.

Note: If you have unsaved changes, Studio saves your modifications before running the verification command.

Performing PCF File Verification Manually

It is also possible to verify PCF files outside of Guidewire Studio through the use of the `gwcc verify-types` command line utility. You can, for example, run this command manually or use the command in an automated test process.

Working in Guidewire Studio

This topic discusses a number of common tasks related to working in Guidewire Studio.

This topic includes:

- “Entering Valid Code” on page 123
- “Using Studio Keyboard Shortcuts” on page 128
- “Viewing Keyboard Shortcuts in ClaimCenter” on page 133
- “Using Text Editing Commands” on page 134
- “Navigating Tables” on page 134
- “Refactoring Gosu Code” on page 134
- “Saving Your Work” on page 136

Entering Valid Code

Guidewire Studio provides several different ways of obtaining information about ClaimCenter objects and APIs to assist you in writing valid rules and Gosu code. These include:

- **Code menu commands** complete code and object names to assist in writing valid Gosu code and in navigating within and around Gosu code.
- **Dot completion** opens a context-sensitive pop-up window that contains all the subobjects and methods that are valid for this object, in this context. Dot completion also works with Gosu or Java packages.
- **SmartHelp** displays a list of valid fields and subobjects for the current object.
- **Gosu API Reference** material provides a searchable reference on the Gosu APIs, methods and properties.
- **PCF Reference Guide** (or *PCF Format Reference*) provides a description of the PCF widgets and their attributes that you can use within the PCF editor.

- **Gosu Reference Guide** provides information on the Gosu language. From this menu link, you can access the entire Guidewire documentation suite.

Note: Keyboard commands provide code completion, code navigation, and code editing shortcuts. See “Using Studio Keyboard Shortcuts” on page 128 for information on keyboard shortcuts.

The Code Menu

The Code menu contains a number of commands that you can use to navigate within and around your Gosu code. The following table lists these commands.

Command	Keyboard shortcut	Description
Go to Type	CTRL+N	Opens the Go To Type popup that enables quick navigation to other types. For example, enter a search string to find a Gosu class. You can perform: <ul style="list-style-type: none">• Simple wildcard searches with multiple asterisks in the search string• Camel-case searching in which you type part or all of an acronym. For example, entering CrA returns a list including CreateActivityPattern and CreateAttribute.
Go to Symbol	ALT+CTRL+SHIFT+N	Opens the Go to Symbol popup, which you can use to quickly navigate to symbols (variables, methods, and similar items.) in all types.
Go to Line	CTRL+G	Opens the Go To Line popup, which you can use to quickly navigate to particular lines in a file.
Go to Overridden Impls	ALT+CTRL+B	Displays a popup of possible implementations, which you can then use to jump to one of them.
Go to Super Impl	CTRL+U	Jumps to the super method, if the method has been overridden. This command works only if you place the caret on a method declaration.
Go to Declaration	CTRL+B	Jumps to the declaration of the symbol at the current point. To use, place the caret in any reference to a class, class member, or local variable and press CTRL+B to go to the corresponding declaration. For example, placing the caret in Sides in the Triangle class and pressing CTRL+B takes you to the corresponding declaration in class Shape. <pre>class Triangle extends Shape { function Triangle() { Sides = 3 } } class Shape { var _sides : int as Sides function Shape() { } }</pre>
Go to Next Error	F2	Moves the caret to the first code error in the editor window. Pressing it again (after correcting the error) moves you to the next error, if there is one.
Complete Value	CTRL+%	Opens a popup for completing the value at the current point (for example, on the right hand side of an assignment to a typekey). To activate, type an object or entity name, then press CTRL+Slash to open the object value selection dialog. For example, typing the following: <code>Activity.AssignedByUser ==</code> and then pressing CTRL+Slash, opens a selection dialog in which you can choose a specific user from the User list.

Command	Keyboard shortcut	Description
Complete Code	CTRL+SPACE	Opens a popup for completing the partial expression at the current point. This includes: <ul style="list-style-type: none">• Completing a partial word.• Completing a property or method on a partially formed expression.• Automatically adding a “uses” statement for a type that has not been imported yet.
Complete Class Name	ALT+CTRL+SPACE	To use, type a portion of an object name, then press CTRL+SPACE. If there are multiple objects that fit the initial letter combination, Studio opens an Objects and Functions popup window in which you can choose the correct object. For example, entering Ac then pressing CTRL+SPACE opens a popup in which you can chose either actions or Activity.
Parameter Info	CTRL+P	Opens a list of resources starting with the initial letters that you typed.

Using Dot Completion

You can use Studio to complete your code by placing the cursor (or caret) at the end of a valid ClaimCenter object or subobject and typing a dot (.). This action causes Studio to open a pop-up window that contains all the subobjects, methods, and properties that are valid for this object, in this context.

As you type, Studio filters this list to include only those choices that match what you have typed thus far. Use the down arrow to highlight the item you want and press **Enter**, the space bar, or **Tab** to select it. After you select an item, Studio inserts it in the correct location in the code.

Dot Completion Icons

The dot completion pop-up window contains all the subobjects, methods, and properties that are valid for this object, in this context. In front of each item is a graphical icon that provides additional information. Using these icons, for example, you can determine if a property (or field) is a native property on the object or if it is derived from an enhancement.

As you look at the icons, you can see, for example:

- Enhancement methods and properties contain a green plus symbol.
- Private methods and properties contain a padlock symbol.
- Protected methods and properties contain a key symbol.
- Internal methods and properties contain a closed letter symbol.
- Database-backed properties (meaning properties on entities that exist in the database) contain a database symbol.

Field icons

In the pop-up window, you see various icons before a property on the list. These icons have the following meanings:

Icon	Meaning
	Public property
	Private property
	Protected property

Icon	Meaning
	Internal property
	Public enhancement property
	Private enhancement property
	Protected enhancement property
	Internal enhancement property
	Database-backed entity property

Method icons

In the pop-up window, you see various icons before a method on the list. These icons have the following meanings:

Icon	Meaning
	Public method
	Private method
	Protected method
	Internal method
	Public enhancement method
	Private enhancement method
	Protected enhancement method
	Internal enhancement method

Gosu and Java Package Name Completion

Dot completion also works with Gosu or Java packages. You can enter a package name and press dot to get a list of packages and types within the package name before the dot. Studio can complete all packages and namespaces within the Studio type system including PCF types. Studio filters the list as you type to include only those choices that match what you have typed thus far.

Using SmartHelp

As you type in a Studio editor, Studio assists you in entering valid code through the use of its **SmartHelp** feature. **SmartHelp** assists you in entering valid entries for method parameters, object types, entity literals, and other entities.

Studio uses a light-bulb icon  to indicate that SmartHelp is available for the current line of code. Clicking the SmartHelp icon gives you one of the following:

Item	Description
A list of valid types	Clicking the icon opens a window that contains a list of valid types from which to choose. If the field contains values from a typelist, Studio shows only the valid values.
A text field for data entry	Clicking the icon opens a text field that you can use to enter the correct type of data, for example, a String or Number entry field. However, Studio can sometimes place quotation marks around a value entered through this method inappropriately. Be especially careful of entering null in an entry field.
A menu of further choices	Clicking the icon provides the following additional choices If Studio is unable to determine your intent: <ul style="list-style-type: none">• Entity Selection opens a dialog box that displays an object hierarchy that you can expand to find the correct object. It can take several seconds for the dialog to open as Studio must build the entity list first.• Search... opens a dialog box in which you can enter criteria to search for the correct entity.

Accessing the Gosu API Reference

Guidewire provides an API reference that you can use to obtain additional information about the Gosu APIs and their methods and parameters. There are two ways to access this information:

- Selecting **Gosu API Reference** from the Help menu opens the full **Guidewire Studio Gosu API Reference** in a separate window. Use the **Search** functionality to find information on an API, or expand the API list and select a field, parameter or method to view.
- Placing the cursor in a method name in a line of code, and then pressing **F1** displays a pop-window with information about that method, including information about its parameters.

The **Gosu API Reference** window contains a search panel, a contents tree, and a display area.

- The contents tree displays a tree view of the type system, organized by package.
- The search panel contains a text field for search terms, a button to clear the text field, a search button, and a re-index button. It also includes an indication of the time of the last indexing operation.

If the Reference has never been indexed, then you must index it before proceeding.

Accessing the PCF Reference Guide

Guidewire provides a PCF reference, *Guidewire ClaimCenter PCF Format Reference*, that you can use to obtain information about PCF widgets and their attributes. To access the reference guide, select **PCF Reference Guide** from the Studio Help menu. Studio opens the guide as a searchable HTML page in a browser window.

Accessing the Gosu Reference Guide

It is also possible to access the Guidewire ClaimCenter documentation suite from within Guidewire Studio. To do so, select **Guidewire Gosu Reference** from the Studio Help menu. Studio opens a searchable version of the entire Guidewire ClaimCenter documentation suite in a browser window.

Using Studio Keyboard Shortcuts

Guidewire Studio provides a number of keyboard commands (keystrokes) that provide important code completion, navigation, and editing capabilities. The following table lists the command categories:

- | | |
|-------------|---|
| Gosu Editor | <ul style="list-style-type: none">• Intelligent Coding Commands• Find Commands• Code Navigation Commands• Refactoring Commands• Help Commands• Code Editing Commands• Testing Commands• General Commands |
| PCF Editor | <ul style="list-style-type: none">• PCF Editing Commands |

Gosu Editor

The following keystroke shortcuts work in the Studio Gosu editor, such as in rules and classes.

Intelligent Coding Commands

Key	Name	Description
CTRL+SPACE	Smart Complete	<p>Opens a popup for completing the partial expression at the current point. This includes:</p> <ul style="list-style-type: none"> • Completing a partial word. • Completing a property or method on a partially formed expression. • Automatically adding a “uses” statement for a type that has not been imported yet. <p>To use, type a portion of an object name, then press CTRL+SPACE. If there are multiple objects that fit the initial letter combination, Studio opens an Objects and Functions popup window in which you can choose the correct object. For example, entering Ac then pressing CTRL+SPACE opens a popup in which you can chose either actions or Activity.</p>
CTRL+ /	Complete Value	<p>Opens a popup for completing the value at the current point (for example, on the right hand side of an assignment to a typekey).</p> <p>To activate, type an object or entity name, then press CTRL+ / to open the object value selection dialog. For example, type the following:</p> <pre>Activity.AssignedByUser ==</pre> <p>Then press CTRL+ / to open a dialog in which you can choose a specific user from the User list.</p>
ALT+ENTER	Smart Fix	Attempts to correct the error nearest the caret.
F2	Go To Next Error	Moves the caret to the first code error in the editor window. Pressing it again (after correcting the error) moves you to the next error if there is one.

Find Commands

Key	Name	Description
CTRL+F	Find	<p>Searches within an open view for the provided text string. You can make the search case sensitive or search on a regular expression by selecting the appropriate check box. For example, searching on a regular expression of \d (digit) highlights 5 in the following Gosu code:</p> <pre>var test = 5</pre>
CTRL+R	Replace	<p>Searches within an open view for a text string and replaces it with the supplied value. You can also make the search case sensitive or search on a regular expression by selecting the appropriate check box.</p>
F3	Find Next	<p>Performs the previous search again without opening up the search dialog. The search algorithm does not repeat a match until it cycles through all other matches in the open view.</p>
ALT+F7	Find Usages	<p>Searches either through the local view or globally (depending on your selection) for usages of an item under the cursor in a statement or expression. For example, placing your cursor in a method signature searches for other usages of that method.</p>
CTRL+SHIFT+F	Find in Path	<p>Searches for a resource within Studio using the provided resource name or text string. For details of this command, see “Using Find-and-Replace” on page 170.</p>
CTRL+SHIFT+R	Replace in Path	<p>Searches for a text string and replace its value with the supplied value. For details of this command, see “Using Find-and-Replace” on page 170.</p>

Find Commands

Key	Name	Description
ALT+F1	Find in Resources	Highlights the file name of the file currently loaded in the active view. You can also access this command in the following ways: <ul style="list-style-type: none">• Right-click the active tab and select Find in Resources.• Select Find Selected View in Resources from the toolbar Window menu.

Code Navigation Commands

Key	Name	Description
CTRL+N	Go To Type	<p>Opens the Go To Type popup that enables quick navigation to other types. For example, enter a search string to find a Gosu class. You can perform:</p> <ul style="list-style-type: none"> Simple wild-card searches with multiple asterisks in the search string. Camel-case searches in which you type part or all of an acronym. For example, entering CrA returns a list including CreateActivityPattern and CreateAttribute. <p>Selecting a type from the list opens the editor view for that type. You can also depress the keyboard Enter key to select the first type in the list.</p> <p>Note You can use either a camel-case search algorithm or a wild-card algorithm search. You cannot, however, use both types in the same search string. For example:</p> <ul style="list-style-type: none"> Entering GL*Enh finds GLCostSetEnhancement. It does not find GeneralLiabilityLineEnhancement. Entering GLL does find GeneralLiabilityLineEnhancement.
CTRL+ALT+SHIFT+N	Go To Symbol	<p>Opens the Go to Symbol popup, which you can use to quickly navigate to symbols (class fields, PCF variables, and similar items) in all types. Selecting a symbol opens the editor view for the type containing the symbol and puts the focus on the symbol in whatever way is appropriate for the editor.</p>
CTRL+E	Go To Recent	Opens the Recent Views popup, which you can use to quickly navigate to recently edited files.
CTRL+G	Go To Line	Opens the Go To Line popup, which you can use to quickly navigate to particular lines in a file.
CTRL+B	Go To Declaration	<p>Jumps to the declaration of the symbol at the current point. To use, place the caret in any reference to a class, class member or local variable and press CTRL+B to go to the corresponding declaration.</p> <p>For example, placing the caret in Sides in the Triangle class and pressing CTRL+B takes you to the corresponding declaration in class Shape.</p> <pre>class Triangle extends Shape { function Triangle() { Sides = 3 } } class Shape { var _sides : int as Sides function Shape() { } }</pre>
CTRL+ALT+B	Go To Overridden Impl	Opens a popup of possible implementations that you can use to jump to one of them.
CTRL+U	Go To Super Impl	Jumps to the superclass, if the method has been overridden. The caret must be on a method declaration for this command to work.
CTRL+H	Show Inheritance Graph	Opens the inheritance graph popup.
CTRL+ALT+H	Show Call Graph	Opens the call graph popup.
CTRL+SHIFT+F7	Highlight Local Usages	Displays all the local usages of the variable at the current point.
CTRL+F12	Show Class Structure	Opens a popup of members of the current class. This is active only if current view is a class.
ALT+LEFT ARROW	Jump Back	Jumps back to the last location that you visited in this editor.
ALT+RIGHT ARROW	Jump Forward	Jumps forward to the location from which you just jumped. (You can also use this in conjunction with ALT+Left Arrow to jump back and forth between two files.)

Refactoring Commands

Key	Name	Description
CTRL+ALT+V	Extract Variable	Extracts a variable from the current selection.

Help Commands

Key	Name	Description
CTRL+F1	Help Window	Opens the Help window.
CTRL+Q or F1	Context Help	Displays documentation for the element at the current point.
CTRL+T	Show Type at Point	Displays the program type at the current point.
CTRL+SHIFT+T	Copy Type at Point	Copies the type of the program at the current point to the clipboard.
CTRL+P	Show Parameter Info	Displays information on the parameter at the current point in a method call.

Code Editing Commands

Key	Name	Description
Tab	Bulk Indent	If a selection exists, indent all the currently selected lines.
SHIFT+Tab	Bulk Unindent	If a selection exists, Unindent all the currently selected lines.
CTRL+D	Duplicate	If a selection exists, duplicate the selection. If none exists, duplicate the current line.
CTRL+W	Expand Selection	Expand the current selection to the next enclosing expression, statement or logical block.
CTRL+SHIFT+W	Narrow Selection	Narrows the current selection to the next enclosing expression, statement or logical block.
CTRL+X	Cut	Cuts the current selection. If none exists, cuts the current line.
CTRL+C	Copy	Copies the current selection. If none exists, copies the current line.
CTRL+SHIFT+V	Show Copy Buffer	Shows the copy buffer dialog, allowing a previous cut or copy to be selected for paste.
CTRL+ALT+SHIFT+V	Paste Java Code	Pastes the Java code (in the clipboard) into the editor, dynamically translating it to Gosu.
CTRL+SHIFT+UP	Move Selection Up	Moves the current selection up intelligently. If none exists, moves the current line.
CTRL+SHIFT+DOWN	Move Selection Down	Moves the current selection down intelligently. If none exists, moves the current line.
CTRL+SHIFT+/-	Comment/Uncomment	Comments or uncomments the selected lines of code.
CTRL+SHIFT+J	Join Lines	Joins the current line with the next line if no selection exists, or joins all lines within the current selection if it does.

Testing Commands

Key	Name	Description
CTRL+F10	Run Current	Run the current configuration.
F9	Debug Current	Debug the current configuration.
CTRL+SHIFT+F10	Run Current Context	Run the test method under the caret or the entire class if caret is not on a method.
CTRL+SHIFT+F9	Debug Current Context	Debug the test method under the caret or the entire class if caret is not on a method.

General Commands

Key	Name	Description
CTRL+SHIFT+F12	Toggle Code Window	Toggles the main coding area between maximized and restored.
CTRL+Mouse wheel	Font size	Increase or decrease the font size

PCF Editor

The following keystroke shortcuts work in the Studio PCF editor.

PCF Editing Commands

Key	Name	Description
CTRL+X	Cut	Cuts the selected widget. Can be pasted into other applications as the underlying XML representation.
CTRL+C	Copy	Copies the selected widget. Can be pasted into other applications as the underlying XML representation.
CTRL+V	Paste	Pastes the widget currently on the clipboard. After activation, the mouse caret changes and the editor highlights the available locations to paste the widget. To complete the paste, click the appropriate location.
CTRL+D	Duplicate	Duplicates the selected widget and all its children.
DELETE	Delete widget	Deletes the selected widget and all its children.
CTRL+drag	Copy widget	Copies the dragged widget to its new location instead of moving it.
ESC	Deselect	Deselects the selected widget.
ALT+[letter]	Edit property	Places the caret in the first property in the properties editor beginning with [letter].
ALT+/_	Widget search	Places the caret in the widget toolbox filter box.
CTRL+G	Go To Line	Opens the Go To Line popup that you can use for quick navigation to the widget at a specified line in the file.
CTRL+/_	Disable/Enable Widget	Toggles whether the editor displays the selected widget. (You can <i>comment out</i> a widget.)

Viewing Keyboard Shortcuts in ClaimCenter

It is possible to view a list of the shortcut keys that are specific to a ClaimCenter screen. It is important to understand that this procedure displays keyboard shortcuts for an individual application screen only and not the application as a whole.

To view keyboard shortcuts defined for a ClaimCenter screen

1. Navigate to the desired application screen.
2. Press ALT-SHIFT-J to open the Guidewire JavaScript Inspector.
3. Enter the following and click **Inspect**:
`window.keyShortcuts.toJSONString()`

The Inspector displays a list of keyboard shortcuts that are specific to that application screen.

Using Text Editing Commands

Use the following menu commands to perform common text-editing actions on the selected rule or text. If desired, you can use conventional Windows keyboard commands to perform these tasks, or use the Studio toolbar icons.

Command	Description	Actions you take
Edit → Undo Edit → Redo	Undoes/repeats the last completed command	Select Undo (CTRL+Z) or Redo (CTRL+Y) from the Edit menu. This command “undoes” (or repeats) the most recently completed command. The undo/redo functionality is context sensitive. For example: <ul style="list-style-type: none">• If you select the Rules tab, undo/redo operates on rule operations like adding or removing a rule, and similar operations.• If you select a specific rule, undo/redo operates on operations within that rule only.• If you select a class editor, undo/redo operates exclusively on that editor. This does not affect any changes made in the Rules tab. Studio permits unlimited undo and redo operations.
Edit → Cut	Deletes the currently selected item and copies it to the Studio clipboard	Select a rule (from the center pane), or text (in the right-pane), then select Cut from the Edit menu. Using this command deletes the selected item from the Studio interface and places it in the Studio clipboard for further use.
Edit → Copy	Copies the currently selected item to the Studio clipboard	Select a rule (from the center pane), or text (in the right-pane), then select Copy from the Edit menu. Using this command places the selected item in the Studio clipboard, but also leaves it available in the Studio interface.
Edit → Paste	Pastes the contents of the Studio clipboard at the caret location	Select a rule (from the center pane), or insert the caret at the desired place (in the right-hand pane), then select Paste from the Edit menu. Studio inserts the contents of the clipboard at the indicated position. Studio continues to insert the same item using the Paste command until you copy a new item to the clipboard.
Edit → Delete	Deletes the currently selected item without copying it to the Studio clipboard.	Select the item, then use this command to remove it completely without copying it to the Studio clipboard.

Navigating Tables

Guidewire provides a number of keyboard shortcuts to aid you in navigating among cells in a table.

Keyboard Shortcuts	Action
Arrow keys (Up / Down / Left / Right)	Move focus between table cells. The Left and Right keys navigate both the selected text and move to the next or previous cell after the caret reaches the ending or beginning of the text.
Tab / SHIFT+Tab	Move focus to the next horizontally adjacent cell, wrapping around to the next lower or higher row as appropriate.
PageUp / PageDown	Jump focus from the current location to the top-most or bottom-most cell in the column.
Home / End	Move the caret to the beginning or end of the selected text. Jump focus to the cell at the beginning or end of the row after the caret reaches the beginning or ending of the text.
CTRL+Home / CTRL+End	Perform in a manner similar to PageUp and PageDown.

Refactoring Gosu Code

Guidewire Studio contains functionality that you can use to refactor resource and type references. These include:

- Renaming a Gosu Resource
- Moving a Gosu Resource

Renaming a Gosu Resource

Using the Studio **Rename** functionality to rename a PCF or Gosu class file changes the relative type name without changing the package. ClaimCenter contains multiple resources that use physical files and for which the file name must match the type name. If you rename one of these resources, Studio also renames the file as well.

To access the rename functionality, first select the resource in the **Resources** tree, then do one of the following:

- Select **Rename Resource** from the **Code** toolbar menu.
- Right-click and select **Rename Resource**.

The **Rename Resource** dialog contains a single text field in which you enter the new relative name.

Performing a rename changes all references to that type to use the new name. For any PCF, rule, or class file that contain a reference to the renamed type, Studio silently opens the file for edit, if it is not open already. (Class files include class, interface, enhancement, and template files.)

IMPORTANT If you revert a file name change (through the source control system), Studio does not automatically rename all newly renamed resources. For example, during the renaming process, it is possible to rename a class file, with this change renaming the resource name in the rule Gosu. If you revert the class file, Studio changes the class file name back to its original file name. Studio *does not* rename the recently changed resource name in the rule Gosu, however. This is the expected behavior, although it is not necessarily intuitive.

Moving a Gosu Resource

Using the Studio **Move** functionality to move a resource can change the fully qualified type name. This can possibly includes changing the package name as well.

- ClaimCenter contains multiple resources that are backed by physical files and for which the file name must match the type name. If moving one of these resources changes the relative type name, then Studio renames the file as well.
- ClaimCenter contains multiple resource files that are backed by physical files and for which the location in the file system must match its package. If moving one of these resources changes the fully qualified type name (excluding the relative part), then Studio moves the file as well.

Moving a Gosu class resource can involve relocating the resource from one folder to different folder. (Guidewire also uses the term *directory root* for a root folder.) Moving a Gosu resource between directory roots moves the file accordingly.

To access the move functionality, first select the resource in the **Resources** tree, then do one of the following:

- Select **Move Resource** from the **Code** toolbar menu.
- Right-click and select **Move Resource**.

The **Move Resource** dialog contains a a text field for the new fully qualified type name and a drop-down to select the directory root.

Performing a move that changes the fully qualified class name also changes all references to that type to use the new fully qualified name. However, references to the type by its relative name do not change if the move does not change the relative type name.

For any resource that contain a reference to the moved type that also needs to change, Studio silently opens the file for edit, if it is not open already.

Saving Your Work

ClaimCenter automatically saves any modifications made in Studio under the following conditions:

- You move between different tabs (views) within Studio.
- The Studio main window loses focus, for example, by moving to another application.

However, you also have the option of performing manual saves by using the **File** menu.

Command	Description
File → Save Changes	Writes any unsaved changes to your local file or SCM system. You can also use the standard keyboard shortcut CTRL+S save your changes.
Toolbar Save icon 	Works the same way that Save Changes and CTRL+S do.

If you have no unsaved changes pending, these commands are unavailable (grayed out).

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 139
- “Working in the Gosu Editor” on page 140

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 175
Entity Names	Represent an entity name as a text string suitable for viewing in the ClaimCenter interface.	“Using the Entity Names Editor” on page 155
Gosu	Create and manage Gosu code used in classes, tests, enhancements, and interfaces.	“Working in the Gosu Editor” on page 140
LOB (Lines of Business)	Define the six special typelists that define the ClaimCenter Lines of Business (LOBs).	“Using the Lines of Business Editor” on page 583
Messaging	Work with messaging plugins.	“Using the Messaging Editor” on page 161
Page Configuration (PCF)	Graphically define and edit page configuration (PCF) files, used to render the ClaimCenter Web interface.	“Using the PCF Editor” on page 337
Plugins	Graphically define, edit and manage Java and Gosu plugins.	“Using the Plugins Editor”, on page 141
Typelist	Define typelists for use in the application.	“Working with Typelists” on page 315
Workflows	Graphically define and edit application workflows.	“Using the Workflow Editor” on page 419

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

- “Working in Guidewire Studio”, on page 123
- “Configuring External Editors” on page 101
- “Classes” on page 169 in the *Gosu Reference Guide*

Using the Plugins Editor

This topic covers ClaimCenter plugins. A plugin is a mini-program that you can invoke to perform some task.

This topic includes:

- “What Are Plugins?” on page 141
- “Working with Plugins” on page 143

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.

You can choose to implement a plugin as either a Gosu or Java class. Guidewire recommends, however, that you implement plugins in Gosu.

Note: For information on how ClaimCenter uses plugins, see “Plugin Overview” on page 101 in the *Integration Guide*.

Plugin Classes

A Guidewire plugin class implements a specific *interface*. Guidewire provides a set of plugin interfaces in the base configuration for this purpose. Within Studio, if you expand the **Plugins** node, you see a *tree* of interfaces, divided by package. This tree shows every plugin interface available in the base configuration.

- If an implementation of an interface already exists, Studio displays  in front of the interface name.
- If it is possible to implement multiple versions of an interface, Studio displays  in front of the interface name.

- If an implementation of an interface does not exist, Studio displays a grayed-out icon  in front of the interface name.
- If Guidewire has deprecated an interface, Studio displays the interface with a line through its name.

It is not possible to add a new plugin interface to your ClaimCenter system. You can, however, modify the underlying code to create a custom modification of the plugin to suit your business needs. Each plugin name is an exact copy of the underlying interface name, except for messaging and encryption plugins (as it is possible to create multiple implementations of these plugin types).

You create, view, and manage plugins through the Studio Plugins editor.

Plugins with Multiple Implementations

In the base configuration, ClaimCenter provides support for multiple implementations of the following types of plugin interfaces:

- | | |
|-------------------|--|
| <i>Messaging</i> | <ul style="list-style-type: none">• MessageReply• MessageRequest• MessageTransport |
| <i>Encryption</i> | <ul style="list-style-type: none">• IEncryption |

Typically, an installation has only a single implementation of an encryption plugin. However, you can, for example, decide to implement a different encryption algorithm (using a different encryption plugin) as part of an upgrade process. In this case, you must retain your old encryption plugin in order to support the upgrade.

To support multiple implementations of encryption plugins, ClaimCenter provides the following configuration parameter:

`CurrentEncryptionPlugin`

Set this configuration parameter to the `EncryptionID` of the encryption plugin currently in use—if you have configured multiple encryption plugins.

- If you do not provide a value for this configuration parameter, then ClaimCenter uses `IEncryption` as the default value.
- 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.

See Also

- For information on the how to configure your database to support encryption, see “Encryption Integration Overview” on page 401 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 406 in the *Integration Guide*.

Startable Plugins

It is possible to register custom code that runs at server start up in the form of a *startable plugin implementation*. You can use this type of 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. For more information, see “Startable Plugins” on page 128 in the *Integration Guide*.

Working with Plugins

You implement a plugin by selecting an interface and choosing either **Implement** (for single-implementation plugins) or **Add implementation** (for multi-implementation plugins, such as messaging plugins).

- If you select a single-implementation plugin interface for which a plugin already exists, Studio does not permit you to create another plugin implementation.
- If you select a single-implementation plugin interface for which a plugin does **not** exist, Studio creates the plugin and automatically names the new plugin with the interface name.
- If you select a multi-implementation plugin interface (such as a messaging interface), Studio prompts you to enter a name. (The name must be unique.)

Note: If you choose a single-implementation plugin interface, for which an associated plugin already exists, Studio does not permit you to create another plugin. You can, however, open the plugin definition in Studio and add or modify plugin parameters.

This process opens the plugin definition within the Studio **Plugins** editor. Studio labels the plugin with one the following:

- with the interface name
- with plugin name, if a messaging plugin

Setting the Plugin Environment

Within the plugin editor, you can set the plugin deployment environment (the **Env** property) and the server ID (the **server** property).

- Use **Env** 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.

Note: For more information on these two properties, see “Reading System Properties in Plugins” on page 127 in the *Integration Guide*. See also “Using the registry Element to Specify Environment Properties” on page 18 in the *System Administration Guide*.

Adding a New Plugin

Click **Add...** and select either Gosu or Java, depending on the plugin class that you plan to implement.

Gosu Plugins

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

Class	Enter the name of the Gosu class that implements this plugin. In the base configuration, Guidewire places all Gosu plugin classes in the following location: <code>Classes.gw.plugin.<package>.impl</code> Although not mandatory, Guidewire recommends that you follow this practice. Enter the full package path to the Gosu class. For example, use <code>gw.plugin.numbergenerator.impl.NumberGenerator</code> for the Gosu <code>INumberGenerator</code> plugin.
-------	---

Java Plugins

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

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.

After creating the plugin, you add parameters to it using the **Parameters Add** functionality. To define a parameter, 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.

Enabling and Disabling a Plugin

You can choose to make a plugin active or inactive using the **Enabled** checkbox. You can, for example, enable the plugin for testing and disable it for production. It is important to understand, however, that you can still access a disabled plugin and call it from code. Enabling or disabling a plugin 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.

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

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.

Which path you choose depends on the business logic that you want to implement. For information on plugins in general, see “Plugin Overview” on page 101 in the *Integration Guide*. For information on creating and deploying a specific plugin type, see the following:

Plugin type	See
Gosu	“Deploying Gosu Plugins” on page 111 in the <i>Integration Guide</i>
Java	“Deploying Java Plugins” on page 115 in the <i>Integration Guide</i>

Working with Web Services

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

This topic includes:

- “What Are Web Services?” on page 145
- “Using the Web Services Editor” on page 145
- “Creating a New Web Service Proxy Endpoint” on page 148

What Are Web Services?

Web Services enable remote procedure calls from one system to another without assuming that the technology on both ends is the same. ClaimCenter exposes its APIs using the platform-neutral and language-neutral web services *SOAP* protocol (originally an acronym for Simple Object Access Protocol). If a SOAP implementation for a programming language exists, then it is a relatively simple task to use web services to connect to APIs or exchange data.

From Gosu, connecting to a remote SOAP service or publishing a new SOAP service is generally very easy due to the native SOAP features built into Gosu.

Using the Web Services Editor

Guidewire provides a **Web Services** editor to create and manage web services within Studio. To create a new web service, select **Web Service** in the Studio Resources pane, right-click and select **New → Web Service**. This 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. (As you cannot change this name after you create it, Guidewire recommends that you choose the name carefully.) You can enter any legal Gosu identifier. For example, a variable name—or to be more

specific, a package name—is acceptable. After setting the name of the web service (and clicking **OK**), the web service dialog opens.

IMPORTANT Guidewire recommends that you take care in selecting your web service name. It is this name that you use to create the SOAP types within Gosu code. You cannot change it after you set it.

The 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 on the local file system, on a local server, or 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 <code>null</code> 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 146 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. It displays the name of the service and the URL value for the service defined in the WSDL. It also provides a means for you to 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.

IMPORTANT You must stop and restart Studio to register your web service.

Adding a New Configuration Environment

Each 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. You may choose to set this parameter, if—for example—you have multiple deployment environments (a test environment and a development environment). In this case, you may want this plugin to be active in only one of these environments.
- The **server** parameter sets a specific server ID. You may choose to set this parameter, if—for example—you implement ClaimCenter in a clustered environment. In this case, you may want this plugin to be active only on a certain machine within the cluster.

Note: For more information on these two properties, see “Reading System Properties in Plugins” on page 127 in the *Integration Guide*. See also “Using the registry Element to Specify Environment Properties” on page 18 in the *System Administration 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 value of zero specifies no time-out. 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 time-out value.)
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 HTPP for the authentication value. If using Windows authentication, see the following "Using Web Services with Windows Authentication" on page 147.

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.

See also

- For information on working with web services in Guidewire ClaimCenter, see "Web Services (SOAP)" on page 25 in the *Integration Guide*.
- For an example of setting up a web service, see "Calling Web Services from Gosu" on page 73 in the *Integration Guide*.

Creating a New Web Service Proxy Endpoint

It is possible to set a proxy override URL. If set, Studio automatically forwards the service to the proxy web server. To illustrate, suppose that you have the following:

Web service parameter	Value
Web Service Name	IUserAPI
Proxy Web Service	http://localhost:8480/cc/soap/IUserAPI?wsdl
Proxy Override	http://remotehost:8000/cc/soap/IUserAPI

The goal of this example is to proxy this web service so that all requests are not sent to

`http://localhost:8480/cc/soap/IUserAPI`

Instead, ClaimCenter forwards the requests to a proxy location

`http://remotehost:8000/cc/soap/IUserAPI`

To create a new web service proxy endpoint

1. Start Studio, if it is not already running.
2. Navigate to the **Web Services** node.
3. Right-click and select **New → Web Service** from the submenu.
4. Enter the web service name. This is the name of the web service as called from code.

Guidewire defines SOAP stubs to an endpoint within Gosu using the following:

`soap.<Web Service Name>.api.<API Name>`

For example, if the web service name is “Test” and the API name is “IUserAPI”, you would enter the following in Gosu code to access the web service:

`soap.Test.api.IUserAPI`

5. In the web services dialog, click **Edit** (next to the **URL** field) and set the URL of the WSDL. Studio uses this URL to reach the web service. For example, to continue the previous example, enter the following:
`http://localhost:8480/cc/soap/IUserAPI?wsdl`
6. Click **Refresh**. This populates the WSDL JavaDoc at the bottom of the screen.
7. In the **Services Definition** area, click **Lookup Services**. In general—with a few exceptions—each WSDL defines a single service. Studio displays the name of the service and its URL as defined in the WSDL (if any). It also displays a read-only **Override URL** field. You enter the URL of your proxy server in this location. Studio automatically forwards the service to the proxy web server.
8. Select the **Enable** checkbox, which makes the **Override URL** field editable. Enter your proxy server URL. For example, enter the following:
`http://remotehost:8000/cc/soap/IUserAPI`

Implementing QuickJump Commands

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

This topic includes:

- “What Is QuickJump?” on page 149
- “Adding a QuickJump Navigation Command” on page 150
- “Checking Permissions on QuickJump Navigation Commands” on page 152

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.)

Note: 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 150 for details.
StaticNavigationCommandRef	Commands that navigate to a page without accepting user-entered parameters. See “Implementing StaticNavigationCommandRef Commands” on page 152.

Type	Use for
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 152.

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 	<p>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.</p> <p>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.</p>
Command Target	<ul style="list-style-type: none"> • StaticNavigationCommandRef • ContextualNavigationCommandRef 	Target page ID.
Command Arguments	<ul style="list-style-type: none"> • StaticNavigationCommandRef 	Comma-separated list of parameters used in the case in which the target page accepts one or more string parameters. (This is not common.)
Context Symbol	<ul style="list-style-type: none"> • ContextualNavigationCommandRef 	Name of a variable on the user's current page.
Context Type	<ul style="list-style-type: none"> • ContextualNavigationCommandRef 	Type of context symbol (variable).

Implementing QuickJumpCommandRef Commands

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

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

Note: 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 152 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 152 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 151.

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 151 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:

- “The Entity Names Editor” on page 155
- “The Variable Table” on page 156
- “The Gosu Text Editor” on page 157
- “Including Data from Subentities” on page 158
- “Entity Name Types” on page 159

The 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** (display names) editor consists of two parts:

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

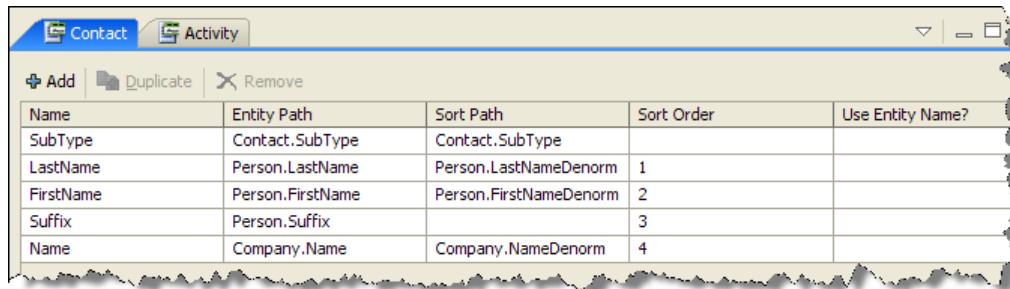
- A Gosu editor that contains the Gosu code that defines the entity name

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

The 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:



The screenshot shows a software interface titled "Contact" with tabs for "Contact" and "Activity". Below the tabs is a toolbar with "Add", "Duplicate", and "Remove" buttons. The main area is a table titled "Variable Table" with the following data:

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

Only use actual columns in the database as members of the Entity Path value. This means that ClaimCenter declares the column in the metadata and that the Data Dictionary does not label that entity column (field) as virtual. 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.

IMPORTANT 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 `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.

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) LastNameDenorm` fields (`Sort Order = 1`). If these values are equal, Studio then compares the values in the `FirstNameDenorm` fields (`Sort Order = 2`), repeating this process for as long as there are fields to compare.

Note: 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.

The 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. This means, for example, that for Contacts, it has to be 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.

IMPORTANT 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:

- “The Messaging Editor” on page 161
- “Working with Email Attachments” on page 164

The 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 148 in the *Integration Guide*
- “Implementing Messaging Plugins” on page 177 in the *Integration Guide*
- “Messaging and Events” on page 139 in the *Integration Guide*

Adding a Message Environment

You can define multiple message environments to suit different purposes. For example, you can set up different message environments for the following:

- A development environment

- A test environment
- A production environment

Guidewire provides a single default message environment in the ClaimCenter base configuration. You see it listed as (Default) in the Env drop-down.

To create a new message environment

1. Select (New) from the Env drop-down list.
2. Enter a name for the message environment.
3. Add message destinations as required. See “Adding a Message Destination” on page 162 for details.

To remove a message environment

1. Select the name of the message environment from the Env drop-down list.
2. Click Remove. ClaimCenter deletes the message environment without asking for confirmation.

IMPORTANT ClaimCenter disables the message environment Remove button if only one message environment exists. However, if there are several message environments, and you have added message destinations to each environment, then there are multiple Remove options available. The Remove button at the top of the screen removes the currently selected message environment. The other Remove option removes the currently selected message destination from the list of destinations. Be careful not to inadvertently click the top Remove 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**, 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** 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 179 in the <i>Integration Guide</i>
Name	The name to use for this messaging destination.

Transport Plugin	<p>The name of the <code>MessageTransport</code> plugin implementation that knows how to send messages for this messaging destination. A destination <i>must</i> define a message transport plugin that sends a <code>Message</code> object over a physical or abstract transport. For example, the plugin might do one of the following:</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 140 in the <i>Integration Guide</i> • “Implementing a Message Transport Plugin” on page 179 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 <i>optionally</i> define a message request (<code>MessageRequest</code>) plugin to prepare or pre-process a <code>Message</code> object before a message is sent to the message transport. For example, the <code>MessageRequest</code> plugin can:</p> <ul style="list-style-type: none"> • Translate strings or codes in a text-type message payload to codes for a remote system. • Translate name/value pairs in a text-type message payload into XML. • Set messaging-specific data model extension properties on the <code>Message</code> object before sending it. <p>To use a message reply plugin, in this Messaging editor field, type the name of the <code>MessageRequest</code> plugin implementation. If the destination requires no special message preparation, omit the request plugin entirely for the destination.</p> <p>For implementation details, see the following:</p> <ul style="list-style-type: none"> • “Implementing a Message Request Plugin” on page 178 in the <i>Integration Guide</i> • “Message Destination Overview” on page 148 in the <i>Integration Guide</i>
Reply Plugin	<p>A destination can <i>optionally</i> define a message reply (<code>MessageReply</code>) plugin to asynchronously acknowledge a <code>Message</code> object. For instance, this plugin can implement a trigger from an external system to notify ClaimCenter that the message send succeeded or failed. To use a message reply plugin, in this Messaging editor field, type the name of the <code>MessageReply</code> plugin implementation. If the destination requires no asynchronous acknowledgement or asynchronous post-processing, omit the reply plugin configuration settings.</p> <p>For implementation details, see the following:</p> <ul style="list-style-type: none"> • “Implementing a Message Reply Plugin” on page 180 in the <i>Integration Guide</i> • “Message Destination Overview” on page 148 in the <i>Integration Guide</i>
Poll Interval	<p>Each messaging destination pulls messages from the database (from the send queue) in batches of messages on the batch server. The application does not query again until this amount of time passes. Use this field to set the value of the polling interval to wait. After the current round of sending, the messaging destination sleeps for the remainder of the poll interval. If the current round of sending takes longer than the poll interval, than the thread does not sleep at all and continues to the next round of querying and sending.</p> <p>For details on how the polling interval works, see the following:</p> <ul style="list-style-type: none"> • “Message Ordering and Multi-Threaded Sending” on page 170 in the <i>Integration Guide</i> <p>IMPORTANT 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>
Initial Retry Interval	The amount of time (in milliseconds) to wait before attempting to retry sending a message after a retryable error condition occurs.
Max Retries	The number of retries to attempt before the retryable error becomes non-retryable.
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.

Field	Description
Number Sender Threads	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. 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. For more information, see the following: <ul style="list-style-type: none">• “Message Ordering and Multi-Threaded Sending” on page 170 in the <i>Integration Guide</i>
Chunk Size	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. For more information, see the following: <ul style="list-style-type: none">• “Message Ordering and Multi-Threaded Sending” on page 170 in the <i>Integration Guide</i>
Shutdown Timeout	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 forbids such actions. The shutdown timeout value is the length of time to wait before attempting to shutdown the messaging subsystem. For more information, see the following: <ul style="list-style-type: none">• “Handling Messaging Destination Suspend, Resume, Shutdown” on page 184 in the <i>Integration Guide</i>.

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** under **Events**. Each event triggers the Event Fired rule set for that destination. Use the special event name wildcard 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 139 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 156 in the *Integration Guide*.

Working with Email Attachments

By default, the `emailMessageTransport` plugin interacts with an external document management system as the *system user* during retrieval of a document attached to an email. It is possible, however, to retrieve the document on behalf of the user who generated the email message instead. To do this, you need to set the `UseMessageCreatorAsUser` in the `emailMessageTransport` plugin.

To set the `UseMessageCreatorAsUser` property

1. In Studio, navigate to the following location:

Configuration → Plugins → gw → plugin → messaging → MessageTransport → emailMessageTransport

2. In the Parameters area, click Add to create a new parameter entry.

3. Enter the following:

Name	UseMessageCreatorAsUser
Value	true

Using the Rules Editor

This topic describes the Studio Rules editor and how you use it to work with Gosu business rules.

This topic includes:

- “Working with Rules” on page 167
- “Changing the Root Entity of a Rule” on page 171
- “Validating Rules and Gosu Code” on page 173
- “Making a Rule Active or Inactive” on page 173

Working with Rules

ClaimCenter organizes and displays rules as a hierarchy within the center pane of Guidewire Studio, with the rule set itself appearing at the root (top level) of the hierarchy tree. Studio displays the **Rule** menu only if you first select a rule set category in the **Resources** pane. Otherwise, it is unavailable.

There are a number of operations involving rules that you can perform in the Studio Rules (Gosu) editor. See the following for details:

- To view or edit a rule
- To change rule order
- To create a new rule set category
- To create a new rule set
- To create a new rule
- To access a rule set from Gosu code
- To find or replace text in the current view
- To find usages of an expression or statement
- To find or replace text throughout the resource tree
- To change the root entity of a rule

To view or edit a rule

1. Expand the **Rule Sets** drop-down list in the left-hand (**Resources**) pane, and then expand the rule set category.
2. Select the rule set to view or edit. All editing and saving within the tool occurs at the level of a rule set.

To change rule order

If you want to change the order of your rules, you can *drag and drop* rules within the hierarchy.

1. Click on the rule that you want to move, hold down the mouse button, and move the pointer to the new location for the rule. Studio then displays a line at the insertion point of the rule. Release the mouse button to paste the rule at that location.
2. To make a rule a child of another rule, select the rule you want to be the child, and then choose **Edit → Cut**. Click on the rule that you want to be the parent, and then choose **Edit → Paste**.
3. To move a rule up a level, drag the rule next to another rule at the desired level in the hierarchy (the reference rule). Notice how far left the insertion line extends.
 - If the line ends before the beginning of the reference rule's name, Studio inserts the rule as a child of the reference rule.
 - If the line extends all the way to the left, Studio inserts the rule as a peer of the reference rule.By moving the cursor slightly up or down, you can indicate whether you want to insert the rule as a child or a peer.

If you move rules using drag and drop, then ClaimCenter moves the chosen rule and all of its children as a group. This makes it easy to reposition entire branches of the hierarchy.

ClaimCenter also supports standard cut, copy, and paste commands, which can be used to move rules within the hierarchy. If you paste a cut or copied rule, Studio inserts the rule as if you added a new rule. It becomes the last child of the currently selected rule.

To create a new rule set category

Guidewire divides the sample rule sets into categories, or large groupings of rules that center around a certain business process, for example, assignment or validation. In the rules hierarchy, rule set categories consist of rule sets, which, in turn, further subdivide into individual rules.

- Rules sets are logical groupings of rules that specific to a business function with Guidewire ClaimCenter.
- Rules contain the Gosu code (condition and action) that perform the actual business logic.

It is possible to create new rule set categories through Guidewire Studio.

1. Select **Rule Sets** in the Studio Resources tree.
2. Right-click and select **New → Rule Set Category** from the menu.
3. Enter a name for the rule set category.

Studio inserts the rule set category in the category list in alphabetic order.

To create a new rule set

In the base configuration, Guidewire provides a number of business rules, divided into rules sets, which organize the business rules by function. It is possible to create new rule sets through Guidewire Studio.

1. Expand the **Rule Sets** node in the Studio Resources tree. Although Guidewire labels the node as rule sets, the node itself actually lists rule set categories.
2. Select a rule set category or create a new rule set category.
3. Right-click and select **New → Rule Set** from the menu.

4. Enter the following information:

Field	Description
Name	Studio displays the rule name in the middle pane, under Rules . For the Guidewire recommendations on rule set names, see “Rule and Rule Set Naming Conventions” on page 33 in the <i>Rules Guide</i> . In general, however, if you create a rule set for a custom entity named Entity_Ext, then you must name your rule set Entity_Ext<RuleSet>. Thus, if you want the custom entity to invoke the Preupdate rules, then name your rule set Entity_ExtPreupdate. There are some variations in how to name a rule set. See the existing rule sets in that category to determine the exact string to append and follow that same pattern with new rule sets in that category.
Description	Studio displays the description in a tab at the right of the Studio if you select the rule set name in the middle pane. Guidewire recommends that you make the description meaningful, especially if you have multiple people working on rule development. In any case, a meaningful rule description is particularly useful as time passes and memories fade.
Entity Type	ClaimCenter uses the entity type as the basis on which to trigger the rules in this rule set. For example, suppose that you select a rule set, then a rule within the set. Right-click and select Complete Code from the menu. Studio displays the entity type around which you base the rule actions and conditions.

To create a new rule

1. Select the rule set to contain the new rule in the Studio Resources pane.
2. Do one of the following:
 - If the new rule is to be a top-level rule, select the rule set name in the middle pane.
 - If the new rule is to be a child rule, expand the rule set hierarchy in the middle pane and select the parent rule.
3. Select **New Rule** from the **Rule** menu, or right-click and select **New Rule**. (You can also press **Ctrl+Shift+N**.)
4. Enter a name for the new rule in the **New Rule** dialog box. Studio creates the new rule as the last child rule of the currently selected rule (or rule set).

To access a rule set from Gosu code

You can access a rule set within a rule set category (and thus, all the rules within the rule set) by using the following Gosu `invoke` method. You can use this method to invoke a rule set in any place that you use Gosu code.

```
rules.RuleSetCategory.RuleSet.invoke(entity)
```

You can only invoke a rule set through the Gosu `invoke` method, not individual rules. Invoking the rule set triggers evaluation of every rule in that rule set, in sequential order. If the conditions for a rule evaluate to true, then ClaimCenter executes the actions for that rule.

Renaming or Deleting a Rule

Use the following menu commands to rename a rule or to delete it entirely from the rule set. You can also access these commands by right-clicking on a rule and selecting the command from the drop-down list.

Command	Description	Actions you take
Rule → Rename	Renames the currently selected rule	Select the rule that to rename in the center pane of Studio, then select Rename from the Rule menu (or right-click and select Rename). Enter the new name for the rule in the Input dialog box. You must save the rule for the change to become permanent.

Command	Description	Actions you take
Edit → Delete	Deletes the currently selected rule	Select the rule to delete in the center pane of Studio, then select Delete from the Edit menu (or right-click and select Delete). (There is no secondary dialog window that asks you to confirm your selection.) You can only use the Delete command to delete rules.

Renaming a rule. At a structural level, Guidewire ClaimCenter stores each rule as a separate Gosu class, with a .gr extension. The name of the Gosu class corresponds to the name of the rule that you see in the Studio Resources tree. ClaimCenter stores the rule definition classes in several different locations, for example:

ClaimCenter/modules/p1/config/rules/...
ClaimCenter/modules/cc/config/rules/...
ClaimCenter/modules/configuration/config/rules/...

If you rename a rule set, ClaimCenter renames the class definition file in the directory structure and any internal class names. It also renames the directory name if the rule has children. Thus, ClaimCenter ensures that the rule class names and the file names are always in synchronization.

Using Find-and-Replace

The Guidewire Studio **Search** menu provides a number of ways that you can search for text or objects in the active Studio resources. You can also access some of the search commands by using the right-click menu of the currently selected resource. Search commands work in one of three modes:

Find/Replace, Find Next	Searches/replaces text in the currently focused view.
Find Usage	Searches for usages of an expression or statement throughout the active resources.
Find/Replace in Path	Searches/replaces text through all or part of the resource tree.

Viewing Search Results

Studio displays the results of a search in a **Search** pane at the bottom of the screen. This pane contains a **Text Filter** entry field that you can use to further refine your search. For example, suppose that the search returns 105 instances of the use of the word *Activity* in a global search. You can further refine your search by entering *Activity.Subject* in the **Text Filter** field. Studio filters the search results and displays (for example) the three occurrences of the search results that contain this specific text.

To find or replace text in the current view

1. Open a rule or class in its Gosu editor. This is the currently selected view.
2. Select either **Find...** or **Replace...** from the **Search** menu.
 - If you chose **Find...**, you can search for text in the currently selected view.
 - If you chose **Replace...**, you can search for and replace text in the currently selected view. To replace text, enter the text string on which to search, and its replacement in the appropriate fields. If Studio finds a text match, it presents you with the option of replacing the text in this one match or in all matches it finds.

Notice that you have several search options from which to choose. These include:

- **Case sensitive** - If checked, Studio searches for an exact text string match, including capitalization.
- **Regular expression** - If checked, Studio uses the provided text string as a search pattern. For example, entering “[Cc]laim” matches to either “claim” or “Claim”, but not to “CLAIM”.

If it finds a match, Studio highlights the text. To move to the next occurrence of the found text, select **Find Next** from the **Search** menu, or press F3.

To find usages of an expression or statement

1. Place the cursor in an expression or statement within the currently selected view. Studio uses the text at the cursor location as the search item. For example, if you place your cursor in the first word in the following expression:

```
Activity.Priority=="high"
```

Studio searches for all usages of the word *Activity*.

2. Select **Find Usages** from the Search menu.

3. Select the scope for the search:

- **Global** - Searches for usages throughout all active resources.
- **Current view** - Searches for usages in the current view only.

Studio displays the results of a search in a **Search** pane at the bottom of the screen.

To find or replace text throughout the resource tree

1. Select a resource.

2. Select either **Find in path...** or **Replace in path...** from the **Search** menu, depending on your purpose.

- If you chose **Find in path...**, then you can search for text or all/part of a resource name.
- If you chose **Replace in path...**, then you can both search for and replace text. To replace text, enter the text string on which to search, and its replacement in the appropriate fields. If Studio finds a text match, it presents you with the option of replacing the text in this one match or in all matches it finds.

Notice that you have multiple options for the search. These include:

- **Search all active resources** - If checked, Studio searches for all active resources, not just the currently selected view.
- **Search in view** - If checked, Studio searches in the current view only.
- **Case sensitive** - If checked, Studio searches for an exact text string match, including capitalization.
- **Regular expression** - If checked, Studio uses the provided text string as a search pattern. For example, entering “[Cc]laim” matches to either “claim” or “Claim”, but not to “CLAIM”.
- **Recursive search** - If unchecked, Studio performs the search in the current node only. If checked Studio performs the search in the current node and all its subnodes.

Studio displays the results of the search in a **Search** pane at the bottom of the screen.

Changing the Root Entity of a Rule

ClaimCenter bases each Gosu rule on a specific business entity. In general, the rule set name reflects this entity. For example, in the Preupdate rule set category, you have **Activity Preupdate** rules and **Contact Preupdate** rules. This indicates that the root entity for each rule set is—respectively—the **Activity** object and the **Contact** object. You can also determine the root entity for a specific rule by using the Studio code completion functionality:

1. Create a sample line of code similar to the following:

```
var test =
```

2. Right-click after the = sign and select **Complete Code** from the menu. The Studio code completion functionality provides the root entity. For example:

```
activity : Activity  
contact : Contact
```

Note: You can also use CTRL-Space to access the Studio code completion functionality.

ClaimCenter provides the ability to change the root entity of a rule through the use of the right-click **Change Root Entity** command on a rule set.

Why Change a Root Entity?

The intent of this command is to enable you to edit a rule that you otherwise cannot open in Studio as the declarations failed to parse. **Guidewire does not recommend the use of this command in other circumstances.**

For example, suppose that you have the following sequence of events:

1. You create a new entity in ClaimCenter, for example, `TestEntity`. Studio creates a `TestEntity.eti` file and places it in the following location:

```
modules/configuration/config/extensions
```

2. You create a new rule set category called `TestEntityRuleSetCategory` in Rule Sets, setting `TestEntity` as the root entity. Studio creates a new folder named `TestEntityRuleSetCategory` and places it in the following location:

```
modules/configuration/config/rules/rules
```

3. You create a new rule set under `TestEntityRuleSetCategory` named `TestEntityRuleSet`. Folder `TestEntityRuleSetCategory` now contains the rule set definition file named `TestEntityRuleSet.grs`. This file contains the following (simplified) Gosu code:

```
@gw.rules.RuleName("TestEntityRuleSet")
class TestEntityRuleSet extends gw.rules.RuleSetBase {
    static function invoke(bean : entity.TestEntity) : gw.rules.ExecutionSession {
        return invoke( new gw.rules.ExecutionSession(), bean )
    }
    ...
}
```

Notice that the rule set definition explicitly invokes the root entity object: `TestEntity`.

4. You create one or more rules in this rule set that use `TestEntity` object, for example, `TestEntityRule`. Studio creates a `TestEntityRule.gr` file that contains the following (simplified) Gosu code:

```
internal class TestEntityRule {
    static function doCondition(testEntity : entity.TestEntity) : boolean {
        return /*start0orule*/true/*end0rule*/
    }
    ...
}
```

Notice that this definition file also references the `TestEntity` object.

5. Because of upgrade or other reasons, you rename your `TestEntity` object to `TestEntityNew` by changing the file name to `TestEntityNew.eti` and updating the entity name in the XML entity definition:

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
        entity="TestEntityNew" ... >
</entity>
```

This action effectively removes the `TestEntity` object from the data model. **This action does not remove references to the entity that currently exist in the rules files.**

6. You update the database by stopping and restarting the application server.

7. You stop and restart Studio.

As Studio reopens, it presents you with an error message dialog. The message states that Studio could not parse the listed rule set files. It is at this point that you can use the **Change Root Entity** command to shift the root entity in the rule files to the new root entity. After you do so, Studio recognizes the new root entity for these rule files.

To change the root entity of a rule

1. Select a rule set.
2. Right-click and select **Change Root Entity** for the drop-down menu. Studio prompts you for an entity name.
3. Enter the name of the new root entity.

After you supply a name:

- Studio performs a check to ensure that the name you supplied is a valid entity name.

- Studio replace occurrences of the old entity in the function declarations of all the files in the rule set with the new entity. This only works, however, if the old root type was in fact an entity.
- Studio changes the name of the entity instance passed into the condition and action of each rule.
- *Studio does not propagate the root entity change to the body of any affected rule. You must correct any code that references the old entity manually.*

IMPORTANT The intent of this command is to enable you to edit a rule that you otherwise cannot open in Studio as the declarations failed to parse. Guidewire does **not** recommend the use of this command in other circumstances.

Validating Rules and Gosu Code

Guidewire Studio provides several means to verify the validity of the rules and Gosu code that you construct. The simplest and most visible is to view the validation status indicators located at the bottom of the right-most Studio pane (or subpanes, for rules). The validation status indicates the following conditions:

- It shows *green* if the code you enter passes the Studio internal validation.
- It shows *yellow* if the code is valid but contains deprecated references.
- It shows *red* if the Gosu code does not pass validation.

If you enter non-valid code, Studio provides an error message next to the validation status indicator.

Making a Rule Active or Inactive

The Rule engine skips any inactive rule, acting as if its conditions are false. (This causes it to skip the child rules of an inactive rule, also.) You can use this mechanism to temporarily disable a rule that is causing incorrect behavior (or that is no longer needed) without deleting it. Sometimes, it is helpful to keep the unused rule around in case you need that rule or something similar to it in the future.

Use the following menu command to make a rule active or inactive.

Command	Description	Actions you take
Rule → Active	Toggles the rule active status	Select a rule in center pane, then select Active from the Rule menu to make it active. Active rules have a double check mark next to Active in the Rule menu besides a check in the box next to the rule name in the center pane. You can also do the following: <ul style="list-style-type: none">• Check the box next to the rule to make it active.• Uncheck the box next to the rule to make it inactive.

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:

- “The Display Keys Editor” on page 175
- “Creating Display Keys in a Gosu Editor” on page 176
- “Retrieving the Value of a Display Key” on page 176

The Display Keys Editor

A `DisplayKey` represents a single user-viewable text string. Guidewire *strongly* recommends that any string constant that can potentially reach the eyes of the user be kept as a `DisplayKey` rather than a hard-coded `String` constant.

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

`ClaimCenter/modules/cc/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. See “Localizing Display Keys” on page 483 for more information.

ClaimCenter represents display keys within a hierarchical name space.

- Within Studio, this translates to a tree structure with expandable nodes.
- Within `display.properties`, this translates into a “.” 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.

Studio does not, however, save changes made in the `Display Keys` editor automatically. You must explicitly save your work. Studio saves your work automatically, however, if you navigate to a different editor.

In Studio, you access the **Display Keys** editor by selecting its node in the **Resources** tree. 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 expanding nodes. To narrow your search, enter a path in the Filter box. For example, if you type <code>Validation.</code> (with the dot), ClaimCenter displays multiple choices for <code>Java.Validation.*</code> .
Modify the text of an existing display key	Select the display key that you want to modify, then modify the string in the text box as desired. (ClaimCenter does not save your work automatically unless you navigate to a different editor.)
Create a new display key	Select the root path of the new display key, then click  to open the Create Display Key dialog. Enter the display key name and default value. To add a new root, open the Create Display Key dialog, delete the existing root, and enter a new root.
Delete an existing display key	Select the display key that you want to delete and click  to remove it.
Localize an existing display key	Select a different locale and enter the localized text. See “ Localizing Display Keys ” on page 483.

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. As you do so, Studio queries you as to your intention. (Studio also displays a wavy line beneath a text string for which you can create a display key.) If you press ALT-Enter, Studio opens the **Create Display Key** dialog, from which you can create a display key directly.

For example, suppose that you enter the following in the **Rule Actions** pane in the Rule editor:

```
var errorString = "SendFailed"
```

Studio displays a wavy line under the “SendFailed” string. If you press ALT-Enter, Studio opens the **Create Display Key** dialog and populates the **Display Key Name** field with the string text that you entered in the Rule editor. The dialog contains a text entry field in which you can enter the localized text for the string that you entered in the Rule editor.

After you enter the text and click **OK**, Studio shows the following in the Rule editor:

```
var errorString = displaykey.SendFailed
```

If you want to enter multiple entries for multiple locales, then select the **Specify values for each locale** checkbox. You can enter a localized string for each locale that exists in Guidewire ClaimCenter.

You can find the new display key by entering its name in the Studio **Display Keys** editor filter.

See Also

- “[Localizing Display Keys](#)” on page 483

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 as a tool to use to understand and visualize 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 181
- “What Can You View in the Data Dictionary?” on page 182
- “Using the Data Dictionary” on page 183

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 32 for instructions on how to regenerate these dictionaries.

What Can You View in the Data Dictionary?

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. 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 (Conversion View)**.

The standard and conversion 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 183 discusses the standard *Data Dictionary* view in more detail.
- Use the *conversion 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 Conversion 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 conversion view of the *Data Dictionary* differs from the standard view in the following respects:

- The conversion view displays subtype fields interspersed with supertype fields. For example:
 - `fieldA`
 - `fieldB` (only for subtype XYX)
 - `fieldC` (only for subtype DFG)
 - `fieldD`
- The conversion view does *not* show virtual fields or virtual arrays.
- The conversion view does *not* show non-loadable columns. For example, it does not show `createUserID` or `createTime`.
- The conversion view *omits* any non-persistent entities.
- The conversion view *omits* entities that are persistent but non-loadable. For example, **Group** is not loadable. Therefore, the conversion 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 315 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:

- Object Attributes
- Entity Subtypes
- Data Column and Field Types
- Virtual Properties on Data Entities

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.

Note: 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 **AccountContactRoleStubypes** 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 examples use the ClaimCenter application 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

This topic describes the data objects (entities) that Guidewire provides as part of the base data model configuration.

This topic includes:

- “What Is the Data Model?” on page 187
- “Overview of Data Entities” on page 189
- “Base ClaimCenter Data Objects” on page 196
- “Data Object Subelements” on page 209

What Is the Data Model?

At its simplest, the Guidewire data model is merely an XML-formatted list of entities and typelists.

Entities	Each entity has a set of fields defined for it. You can add the following kinds of fields to an entity: <ul style="list-style-type: none">• Column• Type key• Array• Foreign key
Typelists	A typelist is a list of predefined code/value pairs (also called type codes) that you can apply to the fields on an entity. There are several different types: <ul style="list-style-type: none">• <i>Internal</i> typelists - You cannot modify internal typelists as the application depends upon them for internal application logic.• <i>Extendable</i> typelists - You can modify this kind of typelist according to its schema definition.• <i>Custom</i> 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 application metadata files on startup. This instantiates the data model as a collection of columns and rows in the database that interact with Java or Gosu objects in the application domain.

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

One trade-off for this configurability is that it does require an extensive understanding of the application's base data model. The base data model specifies the entities, fields, and other definitions that comprise a default installation of ClaimCenter.

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

Guidewire expressly designed ClaimCenter so that you can change the base data model to accommodate your business needs. You define changes to the data model through XML files. ClaimCenter stores these files in the following application directory

`ClaimCenter/modules/configuration/config/extensions`

However, you **always** access the data model files through ClaimCenter Studio.

Guidewire calls these changes *data model extensions*. For example, you can *extend the data model* by adding new fields to the `User` entity, or you can create entirely new entities. The complete ClaimCenter data model consists of a base model combined with any custom data model extensions that you create.

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 sufficient to prevent the application from starting thereafter.

Using Dot Notation

There are many places within ClaimCenter that you need information about fields within the application, especially while you configure ClaimCenter. For example, within Gosu (either a business rule, or class or enhancement), there can be a need to check the *owner* of an assignable object. Or, there can be a 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.

The root object is the starting point for any field reference. It is easiest to understand what a root object is by considering an example. If you run Gosu rules on a claim, 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 uses *dot notation* for relative references. This is mostly likely familiar to people with experience in object-oriented programming languages. For example, assume that your Guidewire application 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 Object Files

WARNING Do **not** modify any of the base data object definitions. You can invalidate your installation and prevent it from starting thereafter. You can view these files (in read-only mode) in Studio in the `Data Model Extensions` folder. Studio displays a dim (less bright) icon for those files that are read-only and which you must never modify. In particular, **never** attempt to modify any file in the `Data Model Extensions → metadata` folder.

To better understand the syntax of the object or entity metadata, it is sometimes helpful to look at the ClaimCenter base data model definitions. ClaimCenter defines each base entity (and extension entity) in its own separate metadata file. The file extension of the metadata file depends on the entity type. The file extension of the metadata file determines its purpose, according to the following table.

Extension	Meaning	Contains	Entity type
.dti	Data Type Info	A single data type definition.	datatype
.eix	Entity Internal eXtension	A single entity internal extension for use by Guidewire only.	internalExtension
.eti	Entity Type Information	A single entity declaration. The name of the file corresponds to the name of the entity.	component delegate deleteEntity entity nonPersistentEntity subtype viewEntity
.etx	Entity Type eXtension	A single entity extension.	extension viewEntityExtension
.tix	Typelist Internal eXtension	A single typelist extension for use by Guidewire only.	internalTypelistExtension
.tti	Typelist Type Info	A single typelist declaration. The name of the file corresponds to the name of the typelist.	typelist
.txx	Typelist Type eXtension	A single typelist extension.	typelistExtension

The following list describes the location of the metadata declaration files.

File type	Location
.dti	cc/config/datatypes pl/config/datatypes
.eix	cc/config/metadata

File type	Location
.eti	cc/config/extensions cc/config/metadata pl/config/metadata
.etx	cc/config/extensions configuration/config/extensions
.tix	cc/config/metadata
.tti	cc/config/extensions cc/config/metadata pl/config/metadata
.txx	cc/config/extensions

Example Activity Files

For example, in the base configuration, ClaimCenter defines the following files, all of which relate to the **Activity** entity:

File	Location	Purpose
Activity.eti	pl/config/metadata	Main entity definition, not modifiable.
Activity.eix	cc/config/metadata	Internal entity extension, not modifiable.
Activity.etx	cc/config/extensions	Guidewire entity extension. You cannot <i>modify</i> this entity definition. You can, however, <i>override</i> certain attributes of the entity definition by creating your own Activity.etx file.

If you extend the **Activity** entity, then you need to create the following extension file through Guidewire Studio.

File	Location	Purpose
Activity.etx	configuration/config/extensions	User-created entity extension. You can both create an entity extension of this type and you can modify an existing one. WARNING Any entity declaration file that you create, you need to create through Guidewire Studio to ensure that it resides in the correct location. If you attempt to manually create entity declaration files and do so incorrectly, you can invalidate your Guidewire installation.

You define data entities through XML elements in the metadata files. The root element of an entity definition specifies the kind of entity, and any attributes that apply. Subelements of the entity define the entity components, such as columns (fields) and foreign keys.

The Metadata Directory

The **metadata** directory contains the data model definitions that make up the Guidewire base data model. This directory is a *multi*-directory. This means that Guidewire merges all versions of a file in all module directories called **metadata** to create the complete file. A **metadata** directory contains the following files types:

- *Declaration* files (those with file extensions *.eti and *.tti) can appear only a single time in any of the **metadata** directories.
- *Internal extension* files (those with file extensions *.eix or *.tix) can appear only a single time in any of the **metadata** directories.

The Extensions Directory

The `configuration/config/extensions` directory contains the data model definitions that make up your extensions to the Guidewire base data model. This directory is a *virtual* directory. This means that Studio uses all the files in the `extensions` directories from all the various modules. However, if a file appears twice, then Studio uses the one in the highest module and ignores all the others. In this way, you can create an extension that overrides a Guidewire base configuration entity extensions.

The ‘extensions.properties’ File

In general, if you change the data model (by creating entity extensions, for example), Guidewire ClaimCenter performs a database upgrade the next time you start the application server. However, it is possible to force ClaimCenter to perform a database upgrade without changing any of the data model .etx files.

The `Data Model → extensions` folder contains an `extensions.properties` file that contains a single numeric property. After a database upgrade, ClaimCenter stores the value of this property in the database. At application server start up, ClaimCenter checks the value of the property in the `extensions` file against the value in the database. If the two values do not match, ClaimCenter performs a database upgrade.

Thus, to force a database upgrade, increment the value in the `extensions` file. This action instructs ClaimCenter to perform a database upgrade.

The use of `extensions.properties` in a development environment is optional. However, in a production environment, it is mandatory that you increment the version number if you make changes to the data model and restart the application server.

See Also

- For information on how Guidewire ClaimCenter creates merged virtual directories and the directory hierarchy in general, see “ClaimCenter Configuration Files” on page 88.
- For a discussion of exactly how Guidewire determines the highest-level module, see “How ClaimCenter Interprets Modules” on page 89.

Working with Data Object Files

In working with data object files, you typically want to perform the following operations:

- Search for an existing data object or entity
- Create a new data object
- Extend an existing data object

The following steps describe these procedures.

To search for an entity

1. Within Guidewire Studio, press CTRL-N. This action opens the `Enter type name` dialog.
2. Enter the name of the object or entity that you want to find. Studio displays a list of matching entries that start with the character string you entered.
3. Pay close attention to the file extension. For example, if you enter `Activity`, Studio displays a number of matches, including `Activity.eti` and `Activity.etx`. These are two very different files. Ensure that you are select the correct file.

To create a new data object

1. Within Guidewire Studio, highlight `Data Model Extensions → extensions`.
2. Right-click and select `New → Other file` from the submenu.

3. Enter the name of the object or entity and enter the .eti file extension. You must add the file extension yourself. Studio does not add it for you.

Studio creates an empty file and places it in the **extensions** folder in Studio. Guidewire ClaimCenter stores this file in the application file structure in the following location.

```
.../modules/configuration/config/extensions
```

To extend an existing data object

1. Within Guidewire Studio, do either of the following:

- a. Open **Data Model Extensions** → **extensions** and select the entity that you wish to extend.
- b. Press CTRL-N and search for the entity that you want to extend.

IMPORTANT You must select the entity name that ends with the .eti file extension.

2. Right-click on the entity that you want to extend and select **Create extension file** from the submenu. Studio creates a skeleton version of the file and displays it in a view tab.

Studio creates the file and places it in the **extensions** folder in Studio. Guidewire ClaimCenter stores this file in the application file structure in the following location.

```
.../modules/configuration/config/extensions
```

Important Reminders

As you work with the data objects, remember the following:

- If you are *extending* an existing entity, always select the .eti version of the entity and then choose **Create new extension**.
- If you are *creating* a new entity, always choose **New** → **Other** file and add the .eti file extension.
- If you are *modifying* an existing entity, use CTRL-N to find the entity and always modify the .etx version in the **extensions** folder.

ClaimCenter Data Objects

ClaimCenter uses metadata XML files to define all entities (data objects). ClaimCenter stores the base data object definitions in **datamodel.xsd**. You **never** need modify this file. You can view a read-only version of this file in the **Data Model Extensions** folder in Studio.

WARNING Do not attempt to modify **datamodel.xsd**. You can invalidate your ClaimCenter installation and prevent it from starting thereafter. You can view a read-only version of this file in Studio in the **Data Model Extensions** folder.

File **datamodel.xsd** defines the following:

- The set of allowable or valid data objects
- The attributes associated with each data object
- The allowable subelements on each data object

All ClaimCenter objects (or entities) must correspond to its definition in **datamodel.xsd**.

Using XML files, Guidewire defines a data entity as a particular data object, as a root element in an XML file that bears its name. For example, Guidewire defines the **Activity** object 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 objects into the application database.

Editing XML Data Object Files

WARNING Do **not** modify any of the base data object definitions. You can invalidate your installation and prevent it from starting thereafter. You can view these files (in read-only mode) in Studio in the **Data Model Extensions** folder. Studio displays a dim (less bright) icon for those files that are read-only and which you must never modify. In particular, **never** attempt to modify any file in the **Data Model Extensions** → **metadata** folder. If you create a new extension file manually, you must place it in the proper **modules/configuration/config/extensions** folder.

It is possible to configure Studio to open .xml files directly in an XML editor that is external to Guidewire Studio. To facilitate this process, Guidewire provides an XML attribute named `xmllns`. The `xmllns` attribute defines the *namespace* for the XML elements in the file. This namespace (analogous to a package in Java) serves to further qualify a class name, primarily to avoid collisions between like-named elements from other schemas.

You can configure many XML editors to associate a namespace with an XSD. However, merely defining the namespace within Guidewire ClaimCenter is not sufficient to inform the XML editor which XSD to use in validating an XML document. You must configure the XML editor manually to associate the namespace with the XSD.

IMPORTANT The `xmllns` attribute is currently optional. However, Guidewire strongly recommends that you add it to your entity and typelist files as Guidewire reserves the right to make this attribute required in the future.

Entity files. Use the following for *entity* files (.eti and .etx):

```
<entity xmllns="http://guidewire.com/datamodel" ...>
```

Typelist files. Use the following for *typelist* files (.tti and .ttx):

```
<typelist xmllns="http://guidewire.com/typelists" ...>
```

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. If the editing tool does not handle UTF-8 characters correctly, it can create errors. For XML files, it is possible to use a different file encoding, as long as you specify it in the XML prolog. For all other files, use UTF-8.

See Also

- “Configuring External Editors” on page 101

Data Objects and the Application Database

Guidewire defines each base data object or entity as a root XML element in the file that bears its name. For example, Guidewire defines the **Activity** data object in **Activity.eti**:

```
<entity xmllns="http://guidewire.com/datamodel"
entity="Activity"
...
type="retireable">
...
</entity>
```

Notice that for the base configuration `Activty` object, Guidewire sets the `type` attribute to `retireable`. This type attribute that determines how Guidewire ClaimCenter manages the data object in the ClaimCenter database. For example:

- If a data object has a `type` value of `versionable`, ClaimCenter stores that object in the database with a specific ID and version number.
- If a data object has a `type` value of `retireable`, ClaimCenter stores that object in the database forever. However, you can *retire* or hide the object so that ClaimCenter does not display the object in the interface.

IMPORTANT For each data object in the base configuration and for each entity or object that you create, ClaimCenter automatically generates a field named `ID` that is of data type `key`. This is the internally managed primary key for the object. Do **not** attempt to create objects of data type `key`. This 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 attribute marked as general use to create or extend an data object. Do **not** attempt to create or extend an entity with a `type` attribute marked as *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> Guidewire recommends that you do not attempt to create an entity with a <code>type</code> attribute of <code>editable</code> .
<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</i> . Use <code>versionable</code> instead.
<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</i> . Use <code>versionable</code> instead.
<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 <code>type</code> attribute of <code>nonkeyable</code>.</i>
<code>purgeable</code>	Internal use	An <code>editable</code> entity that you can purge from the system. You do not delete objects of this type from the database. Rather, you mark them as <code>retired</code> . <i>Guidewire recommends that you do not attempt to create an entity with a <code>type</code> attribute of <code>purgeable</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 base entities are of this type.</p> <p>After ClaimCenter adds an instance of a <code>retireable</code> data object to the database, ClaimCenter never deletes the object. Instead, ClaimCenter retires the object. For example, if you select a <code>retireable</code> object in a list view and then click Delete, ClaimCenter preserves the data in the database. However, ClaimCenter inserts an integer in the <code>Retired</code> column for the row represented by that object. Any non-zero value in the <code>Retired</code> column indicates that ClaimCenter considers the object retired.</p> <p>ClaimCenter automatically creates the following fields for a <code>retireable</code> entity:</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 the <code>editable</code> object, with the addition of <code>Retired</code> property.</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>

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
<code>all</code>	Exposed in Gosu, wherever Gosu is valid, for example, in rules and PCF files
<code>doesNotExist</code>	Not exposed in Gosu
<code>hidden</code>	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 (if you set the value to `hidden` or `doesNotExist`), then Guidewire does not generate constructors for the entity. In essence, this means that 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)
<entity>	Yes	No ¹
1. <entity> 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)
<array>	Yes	Yes
<column>	Yes	Yes
<edgeForeignKey>	Yes	Yes
<foreignkey>	Yes	Yes
<onetooone>	Yes	Yes
<typekey>	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 216.

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> attributes	Description	Default
javaClass	Internal.	None
name	Required.	None

Subelements on <component>

The <component> element contains the following subelements:

<component> subelements	Description
column	See “<column>” on page 212.
foreignkey	See “<foreignkey>” on page 220.
fulldescription	See “<fulldescription>” on page 223.
typekey	See “<typekey>” on page 229.

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 both extend existing delegates (if a delegate is marked as extendable) and 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 merely need to 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 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 can not implement directly. There are some delegates that you cannot implement directly through the use of the `<implementsEntity>` element. They are:

- `Versionable`
- `KeyableBean`
- `Editable`
- `Retireable`

Instead of using the `<implementsEntity>` element, you specify these delegate using the `type` attribute on the entity declaration. The basic syntax looks similar to the following:

```
<entity name="SomeEntity" ... type="SomeDelegate">
```

For example, in the base configuration, the Account entity also implements the `Retirable` delegate by setting the entity `type` attribute to `retireable`.

```
<entity entity="Account" ... type="retireable">
  <implementsEntity name="Validatable"/>
  ...
</entity>
```

Delegates that you cannot implement through extension. Guidewire does not permit you to extend base entities (those that Guidewire provides in the base configuration) with certain delegates. These delegates determine the graph to which an entity belongs. Guidewire sets these delegates carefully in the base configuration and you cannot change them thereafter. These delegates include the following:

- `Extractable`

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 241.
- For a discussion of working with delegates in Gosu classes, see “Using Gosu Composition” on page 195 in the *Gosu Reference Guide*.

Attributes of `<delegate>`

The `<delegate>` element contains the following attributes.

IMPORTANT The `requires` attribute on `<delegate>` is strongly interlinked with the `adapter` attribute on `<implementsEntity>`. See that element discussion for details.

<code><delegate></code> attributes	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>

<delegate> attributes	Description	Default
requires	<p><i>Optional.</i> Specifies an interface for which the <i>implementors</i> of this delegate must provide an implementation. By implementors, Guidewire means those entities that refer to the delegate using <implementsEntity>.</p> <p>IMPORTANT This attribute is inter-related with the adapter attributes on <implementsEntity>.</p> <ul style="list-style-type: none"> If you specify a value for the requires attribute, then the implementors of this delegate must specify a value for the adapter attribute on <implementsEntity>. The value of the adapter attribute must be the name of a type that implements the interface specified by the requires attribute of the associated delegate. If you do not specify a value for the requires attribute, then the implementors must not specify an adapter attribute on <implementsEntity>. 	None

Subelements on <delegate>

The <delegate> element contains the following subelements.

<delegate> subelements	Description
column	See “<column>” on page 212.
datetimeordering	<i>Internal.</i>
foreignkey	See “<foreignkey>” on page 220.
fulldescription	See “<fulldescription>” on page 223.
implementsEntity	See “<implementsEntity>” on page 224.
implementsInterface	See “<implementsInterface>” on page 225.
index	See “<index>” on page 226.
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 229.

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 wish to use polymorphism on class methods. For core ClaimCenter classes defined in Java, it is not possible to 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 (through the use of the delegate `requires` attribute) that all implementors of the delegate implement that interface (through `<implementsEntity>`). This delegate usage permits the use of polymorphism and enables delegate implementations to share common implementations on a common superclass.

Delete Entity Data Objects

You use the `deleteEntity` data object to remove a base configuration extension entity from the base data model. Guidewire defines this object in the data model metadata files as the `<deleteEntity>` XML root element.

Attributes on `<deleteEntity>`

The `<deleteEntity>` element contains the following attributes.

<code><deleteEntity></code> attributes	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 250

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.

Attributes on `<entity>`

The `<entity>` element contains the following attributes.

<code><entity></code> attributes	Description	Default
<code>abstract</code>	If True, it denotes that you cannot create an object of this type (you must create a subtype instead) and that any generated code is abstract.	False

<entity> attributes	Description	Default
admin	Whether the tables declared for this object are administration tables. You can always reference rows in the administration tables from rows in the staging tables.	False
base	<i>Internal.</i> Do not use. (The default is <code>false</code>). Guidewire reserves the right to remove this attribute in a future release.	False
cacheable	<i>Internal.</i> If set to <code>false</code> then Guidewire disallows entities of this type (and all its subtypes) from existing in the global cache.	True
consistentChildren	<i>Internal.</i>	False
desc	A description of the purpose and use of the entity.	None
displayName	<p><i>Optional.</i> Generally, use to create a more human-readable form of the entity name. You can access this name using the following:</p> <pre>entity.DisplayName</pre> <p>ClaimCenter uses this value of the name of the subtype key (if you subtype this entity). If you do not specify a value for this attribute, then ClaimCenter uses the name value of the <code>entity</code> attribute as <code>displayName</code>, instead.</p>	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	If <code>true</code> , the entity is available to the ClaimCenter SOAP APIs.	False
extendable	<i>Internal.</i> If <code>true</code> , it is possible to extend this entity.	True
final	If <code>true</code> , you cannot subtype the entity. If <code>false</code> , you can define subtypes using this entity as the supertype.	True
generateInternallyIfAbsent	<i>Internal.</i>	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 refer to 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 <code>ignoreForEvents</code> to <code>true</code> on an entity that references the non-event-generating entity, then ClaimCenter ignores that link as it determines what entities refer to another entity.</p> <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 Changed 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
instrumentationTable	<i>Internal.</i>	False
javaClass	<i>Internal.</i>	None
loadable	If <code>true</code> , you can load the entity through staging tables.	True
lockable	<i>Internal.</i>	False
overwrittenInStagingTable	<p><i>Internal.</i> If <code>true</code> (and the entity is loadable), it indicates that the loader process auto-populates the staging table during import.</p> <p>IMPORTANT If set to <code>true</code>, do not attempt to populate the table yourself as the loader import process overwrites this table.</p>	False
platform	<i>Internal.</i> Do not use. (The default is <code>false</code>). Guidewire reserves the right to remove this attribute in a future release.	False
priority	The priority of the corresponding subtype key. This value is only meaningful for entities participating in a subtype hierarchy, which can be either the <subtype> entities or the root <entity>.	-1
readOnly	<i>Optional.</i>	None

<entity> attributes	Description	Default
setterScriptability	If you choose to set this, use all. If you do not set this value, then ClaimCenter defaults to all. See "Data Objects and Scriptability" on page 195 for more information.	all
size	<i>Internal.</i> The size of the database table that contains this entity.	Large
superTypeEntity	<i>Obsolete.</i> Do not use. Guidewire reserves the right to remove this attribute in a future release.	None
table	<p><i>Required.</i> The database table name in which ClaimCenter stores the data for this entity. ClaimCenter automatically prepends cc_ (for base entities) or ccx_ (for extension entities) to this name.</p> <p>Guidewire recommends that you use 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 true, 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 189 for a discussion of data entity types.	None
validateOnCommit	<i>Internal.</i> If true, ClaimCenter validates this entity during a commit of a bundle that contains this entity.	True

Subelements on <entity>

The <entity> element contains the following subelements.

<entity> subelements	Description
array	See "<array>" on page 210.
aspect	<i>Internal.</i>
checkconstraint	<i>Internal.</i>
column	See "<column>" on page 212.
componentref	See "<componentref>" on page 216.
customconsistencycheck	<i>Internal.</i>
datetimeordering	<i>Internal.</i>
dbcheckbuilder	<i>Internal.</i>
edgeForeignKey	See "<edgeForeignKey>" on page 217.
events	See "<events>" on page 220.
foreignkey	See "<foreignkey>" on page 220.
fulldescription	See "<fulldescription>" on page 223.
implementsEntity	See "<implementsEntity>" on page 224.
implementsInterface	See "<implementsInterface>" on page 225.
index	See "<index>" on page 226.
jointableconsistencycheck	<i>Internal.</i>
onetoone	See "<onetoone>" on page 227.
remove-index	See "<remove-index>" on page 228.

<entity> subelements	Description
tableAugmenter	<i>Internal.</i>
typekey	See “<typekey>” on page 229.
validatetypekeyinset	<i>Internal.</i>
validatetypekeynotinset	<i>Internal.</i>

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 233.

Attributes on <extension>

The <extension> element contains the following attributes.

<extension> attributes	Description	Default
entityName	<i>Required.</i>	None

Subelements on <extension>

The <extension> element contains the following subelements.

<entity> subelements	Description
array	See “<array>” on page 210.
array-override	Use to override (flip) the value of the triggersValidation attribute of an <array> element definition on a base data object. See “Working with Attribute Overrides” on page 239 for details.
column	See “<column>” on page 212.
column-override	Use to override certain very specific attributes on a base data object. See “Working with Attribute Overrides” on page 239 for details.
componentref	See “<componentref>” on page 216.
description	A description of the purpose and use of the entity.
edgeForeignKey	See “<edgeForeignKey>” on page 217.
foreignkey	See “<foreignkey>” on page 220.
foreignkey-override	Use to override (flip) the value of the triggersValidation attribute of a <foreignkey> element definition on a base data object. See “Working with Attribute Overrides” on page 239 for details.
implementsEntity	See “<implementsEntity>” on page 224.
implementsInterface	See “<implementsInterface>” on page 225.
index	See “<index>” on page 226.
internalonlyfields	<i>Internal.</i>
onetoone	See “<onetoone>” on page 227.
onetoone-override	Use to override (flip) the value of the triggersValidation attribute of an <onetoone> element definition on a base data object. See “Working with Attribute Overrides” on page 239 for details.
remove-index	See “<remove-index>” on page 228.
typekey	See “<typekey>” on page 229.

<entity> subelements	Description
typekey-override	Use to override certain specific attributes (fields) of a <typekey> element definition on a base data object. See "Working with Attribute Overrides" on page 239 for details.

Nonpersistent Entity Data Objects

A `NonPersistentEntity` data object defines a temporary (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. Guidewire defines this object in the data model metadata files as the `<nonPersistentEntity>` XML root element.

IMPORTANT You **cannot** extend a persistent entity with a nonpersistent entity.

Guidewire Recommendations

Guidewire recommends, however, 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 as 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 it 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 301 for information on how to associates data types with object properties using the annotation syntax.

Attributes on `<nonPersistentEntity>`

The `<nonPersistentEntity>` element contains the following attributes.

<nonPersistentEntity> attributes	Description	Default
abstract	If True, it denotes that you cannot create an object of this type (you must create a subtype instead) and that any the generated code is abstract.	False
desc	A description of the purpose and use of the entity.	None
displayName	<i>Optional.</i> Generally, use to create a more human-readable form of the entity name. You can access this name using the following: <code>entity.DisplayName</code> ClaimCenter uses this value of the name of the subtype key (if you subtype this entity). If you do not specify a value for this attribute, then ClaimCenter uses the name value of the <code>entity</code> attribute as <code>displayName</code> , instead.	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	Whether the entity is available to the ClaimCenter SOAP APIs.	False

<nonPersistentEntity>			
attributes	Description		Default
extendable	If true, it is possible to extend this entity.		True
final	If true, you cannot subtype the entity. If false, you can define subtypes using this entity as the supertype.		True
javaClass	<i>Internal.</i>		None
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

Subelements on <nonPersistentEntity>

The <nonPersistentEntity> element contains the following subelements.

<nonPersistentEntity>		
subelements	Description	
array	See “<array>” on page 210.	
aspect	<i>Internal.</i>	
column	See “<column>” on page 212.	
componentref	See “<componentref>” on page 216.	
edgeForeignKey	See “<edgeForeignKey>” on page 217.	
foreignkey	See “<foreignkey>” on page 220.	
fulldescription	See “<fulldescription>” on page 223.	
implementsEntity	See “<implementsEntity>” on page 224.	
implementsInterface	See “<implementsInterface>” on page 225.	
onetoone	See “<onetoone>” on page 227.	
typekey	See “<typekey>” on page 229.	

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.

It is only possible to define a subtype for any entity whose final attribute is set to false. (This means that it is impossible to create a subtype of any entity whose final attribute is set to true). If an entity is non-final, then ClaimCenter automatically creates a Subtype field for it.

Attributes on <subtype>

The <subtype> element contains the following attributes:

<subtype>			
attributes	Description		Default
abstract	If true, it denotes that you cannot create an object of this type. Instead, you must create a subtype.		False
desc	A description of the purpose and use of the subtype.		None

<subtype> attributes	Description	Default
displayName	<i>Optional.</i> Occasionally, there is the desire to identify a given object's subtype in the ClaimCenter interface. Use this attribute to set a specific string to use if you want to display the subtype of an object. The use of this parameter is optional. If you do not specify a value for this parameter, then ClaimCenter displays the name of the entity. The entity name is often not user-friendly. Thus, you can use this value to set a more user-friendly version of the subtype name. See also the description of the displayName attribute at "Entity Data Objects" on page 200.	None
entity	The name of the subtype entity. Use this name to access the entity in data views, rules, and other areas within ClaimCenter.	None
final	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.	False
javaClass	<i>Internal.</i>	
priority	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
readOnly	<i>Optional.</i>	None
supertype	The name of the supertype of this subtype.	None

Subelements on <subtype>

The <subtype> element contains the following subelements.

<subtype> subelements	Description
array	See "<array>" on page 210.
aspect	<i>Internal.</i>
checkconstraint	<i>Internal.</i>
column	See "<column>" on page 212.
customconsistencycheck	<i>Internal.</i>
datetimeordering	<i>Internal.</i>
dbcheckbuilder	<i>Internal.</i>
edgeForeignKey	See "<edgeForeignKey>" on page 217.
events	See "<events>" on page 220.
foreignkey	See "<foreignkey>" on page 220.
fulldescription	See "<fulldescription>" on page 223.
implementsEntity	See "<implementsEntity>" on page 224.
implementsInterface	See "<implementsInterface>" on page 225.
index	See "<index>" on page 226.
jointableconsistencycheck	<i>Internal.</i>
onetoone	See "<onetoone>" on page 227.
tableAugmenter	<i>Internal.</i>
typekey	See "<typekey>" on page 229.
validatetypekeyinset	<i>Internal.</i>
validatetypekeynotinset	<i>Internal.</i>

View Entity Data Objects

A `viewEntity` is a logical view of entity data. You can use it to enhance performance during the viewing of tabular data. A `viewEntity` provides a logical view of data for an entity, plus other entities, of interest to a ListView. A `viewEntity` can also 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` entity called `ActivityView`.

Unlike a standard entity, the `viewEntity` entity type does not have an underlying database table. ClaimCenter does *not* persist view entities to the database. A `viewEntity` cannot create a materialized view in the database. (A materialized view is one in which ClaimCenter caches the results of a database query as a database table.) Rather, a `viewEntity` serves as a way of restricting the amount of data that a database query returns.

A `viewEntity` entity improves the performance of ClaimCenter on frequently used pages that list entities. Like other entities, you can subtype `viewEntity` entities. For example, the `My Activities` page makes use of a `viewEntity` entity—the `ActivityDesktopView`, which is a subtype of the `ActivityView`.

As ClaimCenter can export `viewEntity` entities, it generates SOAP interfaces for them.

Guidewire defines this object in the data model metadata files as the `<viewEntity>` XML root element.

Attributes on `<viewEntity>`

The `<viewEntity>` element contains the following attributes:

<code><viewEntity></code> attributes	Description	Default
<code>abstract</code>	If true, it denotes that you cannot create an object of this type. Instead, you must create a subtype.	<code>False</code>
<code>desc</code>	A description of the purpose and use of the entity.	<code>None</code>
<code>entity</code>	<i>Required.</i> Name of this <code>viewEntity</code> object.	<code>None</code>
<code>exportable</code>	If true, the <code>viewEntity</code> object is available to the ClaimCenter SOAP APIs.	<code>True</code>
<code>extendable</code>	If true, it is possible to extend this entity.	<code>True</code>
<code>final</code>	If true, the entity definition is final and you cannot define any subtypes for it. If <code>false</code> , then you can define a subtype using this entity as the supertype.	<code>True</code>
<code>javaClass</code>	<i>Internal.</i>	<code>None</code>
<code>primaryEntity</code>	<i>Required.</i> The primary entity type for this <code>viewEntity</code> object. The primary entity must be keyable. See “Data Objects and the Application Database” on page 193 for information on keyable entities.	<code>None</code>
<code>showRetiredBeans</code>	Whether to show retired beans in the view.	<code>None</code>
<code>supertypeEntity</code>	The name of supertype of this entity.	<code>None</code>

Subelements on `<viewEntity>`

The `<viewEntity>` elements contain the following subelements:

<code><viewEntity></code> subelements	Description
<code>computedcolumn</code>	Specifies a column that has some type of transformation applied to it during querying from the database. Typical transformations are columns that compute a simple value. Sample calculations include <code>col1 + col2</code> or even aggregate columns <code>SUM(col1)</code> .
<code>computedtypekey</code>	Specifies a typekey that has some type of transformation applied to it during querying from the database.
<code>fulldescription</code>	See the discussion following the table.
<code>viewEntityColumn</code>	Represents a column in a <code>viewEntity</code> table

<viewEntity> subelements	Description
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 <code>viewEntity</code> 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 <code>viewEntity</code> 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 a 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 a SQL group automatically.

View Entity Extension 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 on `<viewEntityExtension>`

The `<viewEntityExtension>` element contains the following attributes:

<viewEntityExtension> attributes	Description	Default
entityName	<i>Required.</i> Name of the view entity to extend.	None

Subelements on <viewEntityExtension>

The <viewEntityExtension> element contains the following subelements:

<viewEntityExtension> subelements	Description
computedcolumn	Specifies a column that has some type of transformation applied to it during querying from the database. Typical transformations are columns that compute a simple value. Sample calculations include col1 + col2 or even aggregate columns SUM(col1).
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	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.

Important Caution

Guidewire *strongly* recommends that you do **not** create a view entity extension (ViewEntityExtension) that causes traversals into revised (effdated) data. This 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

The ClaimCenter data model contains the following subelements that you can use with entities. The list does **not** contain elements that Guidewire reserves for internal use.

- <array>
- <implementsEntity>
- <column>
- <implementsInterface>
- <componentref>
- <index>
- <edgeForeignKey>
- <onetoono>
- <events>
- <remove-index>
- <foreignkey>
- <typekey>
- <fulldescription>

Do **not** use the following entity subelements. Guidewire uses these for internal purposes only. As these entity subelements are for internal use, there is no discussion of them in the following topics.

- <aspect>
- <jointableconsistencycheck>
- <checkconstraint>
- <tableAugmenter>
- <customconsistencycheck>
- <validatetypekeyinset>
- <datetimeordering>
- <validatetypekeynotinset>
- <dbcheckbuilder>

<array>

An array defines a set of additional entities (which are all of the same type) to associate with the main entity. For example, a `Claim` entity includes an array of `Document` entities.

Attributes on <array>

The `<array>` element contains the following attributes:

<array> attributes	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. In that case, the owning table can only have one array field for the edge table. Thus, <code>arrayfield</code> must match the name of the field in the array table that corresponds to the owning row.	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.	False
<code>desc</code>	A description of the purpose and use of the array.	None
<code>exportable</code>	Whether the entity is available to the ClaimCenter SOAP APIs. If true, then ClaimCenter can transmit this column as part of any SOAP argument or result, in which case ClaimCenter transmits only a subset of the containing entity. ClaimCenter ignores this attribute if you do not mark the containing table as exportable. If this field applies to a foreign key field, then ClaimCenter creates a reference to a Data object of the type defined by the referenced table.	True
<code>generateCode</code>	<i>Internal.</i>	true
<code>getterScriptability</code>	If you choose to set this, use <code>a11</code> . If you do not set this value, then ClaimCenter defaults to <code>a11</code> . See “Data Objects and Scriptability” on page 195 for more information.	a11
<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 refer to 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 refer to another entity. <ul style="list-style-type: none"> • At the entity level, the <code>ignoreForEvents</code> attribute means changes to (or addition or removal of) this entity do not cause <code>Changed</code> events to fire for any other entity. • At the column level, the <code>ignoreForEvents</code> attribute means changes to this column do not cause the application to generate events. 	False
<code>name</code>	<i>Required.</i> The name of the property corresponding to this array	None
<code>owner</code>	If true, this entity owns the objects in the array. <ul style="list-style-type: none"> • If you delete 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

<array> attributes	Description	Default
requiredmatch	One of the following values <ul style="list-style-type: none"> • a11—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	If you choose to set this, use a11. If you do not set this value, then ClaimCenter defaults to a11. See “Data Objects and Scriptability” on page 195 for more information.	a11
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 also the discussion on this attribute that follows this table.

The triggersValidation attribute. This attribute—if set to true—can trigger additional ClaimCenter processing. Exactly what it does cause to happen depends on several different factors:

- If the parent entity (the containing entity for the array) is validatable, then any modification to the array triggers the execution of the Preupdate and Validation rules on that entity. (For an entity to be validatable, it must implement the `Validatable` delegate.) This occurs as ClaimCenter attempts to commit a bundle to the database that contains the parent entity.
- 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 it is by default. If this configuration parameter is set to false, ClaimCenter invokes the `IPreUpdateHandler` plugin on the commit bundle instead. ClaimCenter then 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. ClaimCenter does nothing (in the case of `UseOldStylePreUpdate=true`).
 - b. ClaimCenter executes the logic in the `IPreUpdateHandler` plugin on the commit bundle (`UseOldStylePreUpdate=false`).
- In any case, any ClaimCenter processing of the commit bundle excludes the Closed and Reopened validation rules.

Subelements on <array>

The <array> element contains the following subelements:

<array> subelements	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 258 for more information.</p>
fulldescription	See “<fulldescription>” on page 223.
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 256 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 239 for details.

Attributes on <column>

The <column> element contains the following attributes:

<column> attributes	Description	Default
columnName	<p><i>Optional.</i> Name of the corresponding database column. This is different from the name attribute, which specifies the name of the property in the type system. If you do not specify a value for this attribute, then ClaimCenter uses the name value.</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 the following:</p> <ul style="list-style-type: none"> • “maintenance_tools Command” on page 171 in the <i>System Administration Guide</i> • “Working with Attribute Overrides” on page 239 	False
default	Default value given to the field during new entity creation.	None

<column> attributes	Description	Default
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.	False
desc	A description of the purpose and use of the field.	None
exportable	If true, then you can transmit this column as part of a SOAP argument or result. If the containing table is not also marked exportable, then ClaimCenter ignores this attribute.	True
generateCode	<i>Internal.</i>	true
getterScriptability	If you choose to set this, use a11. If you do not set this value, then ClaimCenter defaults to a11. See “Data Objects and Scriptability” on page 195 for more information.	a11
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 refer to 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 refer to 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	The name of the property on the entity. Also the name of the database column, unless you specify the columnName attribute. Use this name to access the column in data views, rules, and other areas within ClaimCenter.	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), 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 as the loader import process overwrites this table.	False
scalable	Whether this value scales as the effective and expired dates change. It only applies to number-type values. (For example, you cannot scale a varchar.) Also, it only applies to effective dated types.	False
setterScriptability	If you choose to set this, use a11. If you do not set this value, then ClaimCenter defaults to a11. See “Data Objects and Scriptability” on page 195 for more information.	a11
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
supportsLinguisticSearch	Applies only to columns of varchar-based data types. <ul style="list-style-type: none"> • If true, searches performed on this field are linguistic. • If false, searches are binary. 	False

<column> attributes	Description	Default
type	<p><i>Required.</i> Data type of the column (field). In the base configuration, Guidewire defines a number of data types and stores their metadata definition files (*.dti files) 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 299. For a list of the data types that you can modify or customize, see “Customizing Base Configuration Data Types” on page 303. For information on how to define new data types, see “Defining a New Data Type: Required Steps” on page 310. 	None

Subelements on <column>

The <column> element contains the following subelements:

<column> subelements	Description	Default
columnParam	See “The <columnParam> Subelement” on page 214.	None
fulldescription	See “<fulldescription>” on page 223.	None
localization	See “The <localization> Subelement” on page 216.	None

The <columnParam> Subelement

You use the <columnParam> element to set the parameters that the data type of the column requires. It is the data type of the column that determines which parameters you can set (or modify) on the column using the <columnParam> element. From the type attribute, you can determine the list of parameters by looking at the definition of the type—the definition of the type as defined in its .dti file.

For example, if you have a mediumtext column, then you can determine the valid parameters for that column by examining file mediumtext.dti. This file indicates that you can modify the following attributes on a mediumtext column:

- encryption
- logicalSize
- trimwhitespace

- **validator**

As 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:

- **modules/pl/config/datatypes** for system-level data types
- **modules/cc/config/datatypes** for optional application-specific data types

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>
```

Which Parameters Can You Define Using `<columnParam>`?

The following list describes the parameters that you can define using `<columnParam>`. These parameters are valid with many—but not all—of the data types.

Parameter	Description
encryption	Whether ClaimCenter stores this column in encrypted format. This only applies to text-based columns. Guidewire allows indexes on encrypted columns (fields). However, as Guidewire stores encrypted fields as encrypted in the database, you must encrypt the input string and search for an exact match to it.
logicalSize	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 (for example, CLOB objects). If you specify a value for the size parameter, then the logicalSize value must be less than or equal to the value of that parameter.
precision	The precision of the field. This applies only if the data type of the field allows a precision attribute. (Precision is the total number of digits in the number.)
scale	The scale of the field. This applies only if the data type of the field allows a scale attribute. (Scale is the number of digits to the right of the decimal point.)
size	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 (as opposed to bytes) the column can hold. WARNING The database upgrade utility automatically detects if the definition of a column has lengthened or shortened the column. If shortening (narrowing) the column, the utility assumes that the instigator of the change has written a version check or otherwise verified that the change does not truncate any data. For both Oracle and SQL Server, if narrowing a column causes data to be lost, the ALTER TABLE statement fails and the upgrade fails.
trimwhitespace	Applies to text-based data types. If true, then ClaimCenter automatically removes leading and trailing white space from the data value.
validator	The name of a ValidatorDef in <code>fieldvalidators.xml</code> . See “ <code><ValidatorDef></code> ” on page 295.

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.

Parameter	Use with data type	Description
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 240 for an example of using a nested `<columnParam>` subelement within a `<column-override>` element to set the `encryption` attribute on a column.

The `<localization>` Subelement

The `<localization>` subelement has one attribute, `tableName`, which is the table name of the localization join table. See “Localized Entities” on page 502 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 on `<componentref>`

The `<componentref>` element contains the following attributes:

<code><componentref></code> attributes	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.	False
<code>desc</code>	A description of the purpose and use of the array.	None
<code>exportable</code>	Whether the entity is available to the ClaimCenter SOAP APIs. If true, then ClaimCenter can transmit this component as part of a SOAP argument or result. ClaimCenter ignores this attribute if you do not mark the containing table as exportable.	True
<code>flatten</code>	Whether to expose a single property representing the <code><component></code> (false), or to expose the individual fields included by the component (true). For example, this attribute controls whether an entity including a component called MoneyComponent has a property that returned the MoneyComponent, or two properties for the Amount and Currency.	False
<code>generateCode</code>	<i>Internal.</i>	true
<code>getterScriptability</code>	If you choose to set this, use all. If you do not set this value, then ClaimCenter defaults to all. See “Data Objects and Scriptability” on page 195 for more information.	all
<code>name</code>	<i>Required.</i> (Although required, it is only important if the value of flatten is false.) Specifies the name of the property on the entity.	None

<componentref> attributes	Description	Default
prefix	An optional prefix to use if defining properties related to the included component. You must use a prefix if you want to include the same component twice on the same entity.	None
ref	<i>Required.</i> The name of the component being included (for example, MoneyComponent).	None
setterScriptability	If you choose to set this, use all. If you do not set this value, then ClaimCenter defaults to all. See “Data Objects and Scriptability” on page 195 for more information.	all

Subelements on <componentref>

The <componentref> element contains the following subelements:

<componentref> subelements	Attributes	Description	Default
annotation	• name • value	Creates a name and value pair. The subelement must contain both attributes.	None
fulldescription	None	See “<fulldescription>” on page 223.	

<edgeForeignKey>

You use the <edgeForeignKey> element in a similar manner to the <foreignkey> element, to define a reference to another entity. Guidewire defines this element in the data model metadata files as the <edgeForeignKey> XML subelement.

Edge foreign keys, however, 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 as 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.

As 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.)

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.

Note: 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.

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 on `<edgeForeignKey>`

The `<edgeForeignKey>` element contains the following attributes.

<code><edgeForeignKey></code> attributes	Description	Default
<code>createhistogram</code>	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 <code><column-override></code> 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 <code>maintenance_tools</code> command. See also the following: <ul style="list-style-type: none"> • “<code>maintenance_tools</code> Command” on page 171 in the <i>System Administration Guide</i> • “Working with Attribute Overrides” on page 239 	<code>False</code>
<code>deletefk</code>	<i>Obsolete.</i> Do not use.	<code>False</code>
<code>deprecated</code>	If <code>true</code> , 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.	<code>False</code>
<code>desc</code>	A description of the purpose and use of the edge foreign key.	<code>None</code>
<code>edgeTableName</code>	The name of the edge table entity. If you do not specify one, then ClaimCenter creates one automatically.	<code>None</code>
<code>edgeTableName</code>	<i>Required.</i> The name of the edge (join array) table to create.	<code>None</code>
<code>exportable</code>	Whether the column is available to the ClaimCenter SOAP APIs. ClaimCenter ignores this attribute if the containing entity is not also marked as exportable.	<code>True</code>
<code>exportasid</code>	If specified, ClaimCenter exposes the field in SOAP APIs as a string, whose value represents the <code>PublicID</code> of the referenced object.	<code>False</code>
<code>fkentity</code>	<i>Required.</i> The entity to which this foreign key points.	<code>None</code>
<code>generateCode</code>	<i>Internal.</i>	<code>true</code>
<code>getterScriptability</code>	If you choose to set this, use <code>a11</code> . If you do not set this value, then ClaimCenter defaults to <code>a11</code> . See “Data Objects and Scriptability” on page 195 for more information.	<code>all</code>

<code><edgeForeignKey></code> attributes	Description	Default
<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 refer to 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 <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 refer to 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 Changed events to fire for any other entity.• At the column level, the <code>ignoreForEvents</code> attribute means changes to this column do not cause the application to generate events.	False
<code>importableagainstexistingobject</code>	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
<code>loadable</code>	If true, then ClaimCenter creates a staging table for the edge table.	False
<code>name</code>	Specifies the name of the property on the entity.	None
<code>nullok</code>	Whether the column can contain null values. This value is meaningless for <code>edgeForeignKey</code> objects.	True
<code>onDelete</code>	<i>Only affects purge actions.</i> If you delete the object referenced by the foreign key, this attribute indicates the action to take on this object <ul style="list-style-type: none">• cascade—Also delete this object.• setnull—Set the foreign key on this object to null (if nullable).• noaction—Do nothing.	cascade
<code>overwrittenInStagingTable</code>	<i>Internal.</i> If true (and the edge 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 as the loader import process overwrites this table.	False
<code>setterScriptability</code>	If you choose to set this, use a11. If you do not set this value, then ClaimCenter defaults to a11. See “Data Objects and Scriptability” on page 195 for more information.	all
<code>soapnullok</code>	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 <code>nullok</code> .	None

Subelements on `<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.

<code><edgeForeignKey></code> subelements	Attributes	Description
<code>fulldescription</code>	None	See “ <code><fulldescription></code> ” on page 223.
<code>implementsEntity</code>	<ul style="list-style-type: none"> • <code>adapter</code> - Interrelated with the <code>requires</code> attribute on <code><delegate></code> • <code>name</code> - name of delegate entity to implement (<code>required = true</code>) 	Applies to the edge table type created by the <code><edgeForeignKey></code> . See also the description for the <code>requires</code> attribute for “Delegate Data Objects” on page 197.

See Also

- “<implementsEntity>” on page 224
- “Delegate Data Objects” on page 197

<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 events to a base entity that does not have events associated with it.

Note: This element is not valid for a `nonPersistentEntity`.

Guidewire defines this element in the data model metadata files as the <events> XML subelement. There can be at most one <events> element in an entity. However, you can specify additional events through the use of <event> subelements. For example:

```
<events>
  <event>
    ...
  </events>
```

Attributes on <events>

There are no attributes on the <events> element.

Subelements on <events>

The <events> element contains the following subelements.

<events> subelements	Description
event	Defines an additional event to fire for the entity. Use multiple <event> elements to specify multiple events. This subelement contains the following attributes: <ul style="list-style-type: none">• <code>description</code> (<code>required = true</code>)• <code>name</code> (<code>required = true</code>) The attributes are self-explanatory. The <event> element requires each one.

<foreignkey>

The <foreignkey> element defines a foreign key reference to another entity.

Attributes on <foreignkey>

The <foreignkey> element contains the following attributes.

<foreignkey> attributes	Description	Default
<code>columnName</code>	<i>Optional.</i> Name of the corresponding database column. This is different from the <code>name</code> attribute, which specifies the name of the property in the type system. If you do not specify a value for this attribute, then ClaimCenter uses the <code>name</code> value. IMPORTANT A common (and recommended) practice is to use the suffix <code>ID</code> for the column name. For example, for a foreign key with name <code>Claim</code> , set the <code>columnName</code> to <code>ClaimID</code> . While Guidewire does not strictly require this, Guidewire strongly recommends that you adopt this practice as it is a way to help in analyzing the database and identify foreign keys.	None

<foreignkey> attributes	Description	Default
createConstraint	If true, the database creates a foreign key constraint for this foreign key.	True
createbackingindex	If true, the database automatically creates a backing index on the foreign key. If set to false, the database does not create a backing index. See “Attribute <code>createbackingindex</code> ” on page 223 for more information.	True
createhistogram	Whether to create a histogram on the column during an update to the database statistics. Note: It is possible to override this attribute on an existing column in an extension (*.etx) file using the <code><column-override></code> 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 <code>maintenance_tools</code> command. See also the following: <ul style="list-style-type: none">• “<code>maintenance_tools</code> Command” on page 171 in the <i>System Administration Guide</i>• “Working with Attribute Overrides” on page 239	False
deletefk	Obsolete. Do not use.	False
deprecated	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.	False
desc	A description of the purpose and use of the field.	None
existingreferencesallowed	If both of the following are set to <code>false</code> (which is not the default): <ul style="list-style-type: none">• <code>loadable</code>• <code>importableagainstexistingobject</code> then, the value in the staging table can only be a reference to an existing object. That is, if there are no rows in the matching source table that reference that same object.	True
exportable	Whether the field is available to the ClaimCenter SOAP APIs. ClaimCenter ignores this attribute if you do not also mark the containing entity as exportable.	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	Required. The entity to which this foreign key refers.	None
generateCode	<i>Internal.</i>	True
getterScriptability	If you choose to set this, use <code>a11</code> . If you do not set this value, then ClaimCenter defaults to <code>a11</code> . See “Data Objects and Scriptability” on page 195 for more information.	<code>a11</code>

<foreignkey> attributes	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 refer to 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 refer to another entity. <ul style="list-style-type: none">• At the entity level, the ignoreForEvents attribute means changes to (or addition or removal of) this entity do not cause Changed events to fire for any other entity.• At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events.	False
importableagainstexistingobject	If true and the entity is importable (loadable), then the value in the staging table can be a reference to an existing object. (This is the publicID of a row in the source table for the referenced object.)	True
includeIdInIndex	If true, then include the ID as the last column in the backing index for the foreign key. This is useful if the access pattern in one or more important queries is to join to this table through the foreign key. You can then use the ID to probe into a referencing table. The only columns that you need to access from the table are this foreign key, and the retired and ID columns. In that case, adding the ID column to the index creates a covering index and eliminates the need to access the table.	False
loadable	If true, you can load the field through staging tables. A staging table can contain a column for the public ID of the referenced entity.	True
name	Specifies the name of the property on the entity.	None
nullok	Whether the field can contain null values.	True
onDelete	<i>Only affects purge actions.</i> If you delete the object referenced by the foreign key, this indicates the action to take upon this object. It can take one of the following values: <ul style="list-style-type: none">• cascade—Also delete this object.• setnull—Set the foreign key on this object to null (if nullable).• noaction—Do nothing.	cascade
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 as 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	If you choose to set this, use a11. If you do not set this value, then ClaimCenter defaults to a11. See “Data Objects and Scriptability” on page 195 for more information.	a11
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 that you want to create a unique, single column (nullable) index on a foreign key. To do this, 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"
        onDelete="noaction"
        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 on <foreignkey>

The <foreignkey> element contains the following subelements.

<foreignkey> subelements	Attributes	Description
fulldescription	None	See “<fulldescription>” on page 223.

<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>

WARNING Any entity that is part of the domain graph **must** implement the Extractable delegate using the <implementsEntity> element. This means 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 add it manually, *the application server refuses to start*.

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.

The <implementsEntity> subelement on any data object definition indicates that the object implements the specified delegate. Guidewire defines this element in the data model metadata files as the <implementsEntity> XML subelement. Guidewire defines any entity that refers to a delegate using <implementsEntity> as an *implementor* of that delegate.

To illustrate, in the PolicyCenter base configuration, Guidewire defines a BACost entity with a name="Cost" attribute on <implementsEntity>. Thus, the BACost *entity* is an implementor of the Cost *delegate*.

```
<?xml version="1.0"?>
<entity ... entity="BACost" ... >
  ...
  <implementsEntity adapter="gw.lob.ba.financials.BACostAdapter" name="Cost"/>
  ...
</entity>
```

In the PolicyCenter base configuration, Guidewire defines a Cost delegate as follows:

```
<?xml version="1.0"?>
<delegate ... name="Cost" requires="gw.api.domain.financials.CostAdapter">
  ...
</delegate>
```

If you provide a value for the optional *requires* attribute on <delegate>, then you **must** provide a value for the *adapter* attribute on <implementsEntity> as well. This value must be the name of a type that implements the interface specified by the *requires* attribute of the associated delegate (in this case, the Cost delegate).

Thus, the BACost entity is an implementor of the Cost delegate as specified by the name attribute on <implementsEntity>. Because the Cost delegate requires an implementation of the CostAdapter interface, the BACost entity must also specify a class that implements this interface in the *adapter* attribute on <implementsEntity>. The following table describes these relationships.

Type	Defines	In file
Entity	<implementsEntity adapter="gw.lob.ba.financials.BACostAdapter" name="Cost"/>	BACost.eti
Delegate	<delegate ... name="Cost" requires="gw.api.domain.financials.CostAdapter"	Cost.eti
Gosu class	class BACostAdapter implements CostAdapter	BACostAdapter.gs

The rule is this:

- If you specify a value for the *requires* attribute on <delegate>, then the implementors of this delegate **must** specify a value for the *adapter* attribute on <implementsEntity>. The value of the *adapter* attribute must be

the name of a type that implements the interface specified by the `requires` attribute of the associated delegate.

- If you do **not** specify a value for the `requires` attribute on `<delegate>`, then the implementors must **not** specify an `adapter` attribute on `<implementsEntity>`.

Attributes on `<implementsEntity>`

The `<implementsEntity>` element contains the following attributes.

<code><implementsEntity></code> subelements	Description
<code>adapter</code>	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.
<code>name</code>	<i>Required.</i> The name of the delegate that this entity must implement.

Subelements on `<implementsEntity>`

There are no subelements on the `<implementsEntity>` subelement.

`<implementsInterface>`

The `<implementsInterface>` subelement on any element indicates that the data entity implements the specified interface. Guidewire defines this element in the data model metadata files as the `<implementsInterface>` XML subelement. This element defines two attributes, an `iface` (interface) attribute and an `impl` (implementation) attribute. If you use this element, then you must specify both of these attributes.

To illustrate, in the base PolicyCenter configuration, Guidewire 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>
```

Interface `BACostMethods` lists a number of getter methods for which any class that implements this interface must provide definitions. These include property getter methods on coverage, state, and vehicle. Defining these methods on `BACost` enables you to use these getter methods in Gosu code.

```
var cost      : BACost
var cov       = cost.Coverage
var state     = cost.State
var vehicle   = cost.Vehicle
```

Attributes on `<implementsInterface>`

The `<implementsInterface>` element contains the following attributes.

<code><implementsInterface></code> subelements	Description
<code>iface</code>	<i>Required.</i> The name of the interface that this data object must implement.
<code>impl</code>	<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. Thus, Guidewire recommends that you add a database index with caution.

Attributes on <index>

The <index> element contains the following attributes.

<index> attributes	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 true, 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 true, 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.	<code>None</code>
<code>trackUsage</code>	If true, 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 true, then ClaimCenter runs an integrity check for unique indexes before loading data from the staging tables.	<code>True</code>

Subelements on <index>

The <index> element contains the following subelements.

<index> subelements	Description	Default
forceindex	<p>Use to force ClaimCenter to create an index if running against a particular database. This is useful as 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 SQL Server database. 	None
indexcol	<p><i>Required.</i> (One or more) Defines a field that is part of the index. Multiple <indexcol> elements specify multiple columns in the same index. 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 (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, suppose that entity A has a foreign key to entity B, and 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 on <onetoone>

The <onetoone> element contains the following attributes.

<onetoone> attributes	Description	Default
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.	False
	If you deprecate an item, use the description to explain why.	
desc	A description of the purpose and use of the field.	None
exportable	Whether the field is available to the ClaimCenter SOAP APIs. ClaimCenter ignores this attribute if you do not also mark the containing entity as exportable.	True
fkentity	<i>Required.</i> The entity to which this foreign key points.	
generateCode	<i>Internal.</i>	True

<onetoone> attributes	Description	Default
getterScriptability	If you choose to set this, use all. If you do not set this value, then ClaimCenter defaults to all. See “Data Objects and Scriptability” on page 195 for more 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 refer to X. If ClaimCenter finds any of these event-generating entity instances, it generates Changed events for those entity instances.	False
	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 refer to 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. 	
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	<i>Required.</i> The name of the field. Use this name to access the field in data views, rules, and other areas within ClaimCenter. The first character of the name must be a letter.	True
setterScriptability	If you choose to set this, use all. If you do not set this value, then ClaimCenter defaults to all. See “Data Objects and Scriptability” on page 195 for more information.	all
triggersValidation	Whether changes to the entity pointed to by this entity trigger validation.	False

Subelements on <onetoone>

The <onetoone> element contains the following subelements.

<onetoone> subelements	Description	Default
fulldescription	See “<fulldescription>” on page 223.	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 (`unique="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`, it is safe to remove this index as 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 on `<remove-index>`

The `<remove-index>` element contains the following attributes.

<code><remove-index></code> attributes	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 315.

Attributes on `<typekey>`

The `<typekey>` element contains the following attributes.

<code><typekey></code> attributes	Description	Default
<code>columnName</code>	<i>Optional.</i> Name of the corresponding database column. This is different from the <code>name</code> attribute, which specifies the name of the property in the type system. If you do not specify a value for this attribute, then ClaimCenter uses the <code>name</code> value.	None
<code>createhistogram</code>	<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 <code><column-override></code> 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 <code>maintenance_tools</code> command.</p> <p>See also the following:</p> <ul style="list-style-type: none"> “<code>maintenance_tools</code> Command” on page 171 in the <i>System Administration Guide</i> “Working with Attribute Overrides” on page 239 	False

<typekey> attributes	Description	Default
default	The default value given to the field during new entity creation.	None
deprecated	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. If you deprecate a typekey, use the description to explain why.	False
desc	A description of the purpose and use of the field.	None
exportable	Whether the field is available to the ClaimCenter SOAP APIs. ClaimCenter ignores this attribute if you do not also mark the containing entity as exportable.	True
generateCode	<i>Internal.</i>	True
getterScriptability	If you choose to set this, use a11. If you do not set this value, then ClaimCenter defaults to a11. See “Data Objects and Scriptability” on page 195 for more 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
nulllok	Whether the field can contain null values.	True
overwrittenInStagingTable	<i>Internal.</i> If true (and the typekey is loadable), it indicates that the loader process auto-populates the typekey in the staging table during import. IMPORTANT If set to true, do not attempt to populate the typekey yourself as the loader import process overwrites this typekey.	False
setterScriptability	If you choose to set this, use a11. If you do not set this value, then ClaimCenter defaults to a11. See “Data Objects and Scriptability” on page 195 for more 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 nulllok.	None
typefilter	The name of a filter associated with the typelist. See “Static Filters” on page 325 for additional information.	None
typelist	<i>Required.</i> The name of the typelist from which this field gets its value. See also “Working with Typelists” on page 315.	None

Subelements on <typekey>

The <typekey> element contains the following subelements.

<typekey> subelements	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 330 for additional information.	None
fulldescription	See “<fulldescription>” on page 223.	None

Subelements on <keyfilters>

The <keyfilters> element contains the following subelements.

<keyfilters> subelements	Description	Default
<keyfilter>	Specifies a keyfilter to use to filter the typelist. This element requires the <name> attribute. 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.	None

Note: You can include multiple <keyfilter> elements to specify multiple keyfilters.

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 233
- “Defining a New Data Entity” on page 237
- “Extending a Base Configuration Entity” on page 238
- “Working with Attribute Overrides” on page 239
- “Extending the Base Data Model: Examples” on page 241
- “Removing Objects from the Base Configuration Data Model” on page 250
- “Deploying Data Model Changes to the Application Server” on page 254

Planning Changes to the Base Data Model

Before proceeding to modify the base data model, Guidewire strongly recommends that you first review the *Guidewire 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.

- Create a new entity with custom fields using any of the entity types listed in “Base ClaimCenter Data Objects” on page 196.
- 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.

IMPORTANT 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 184 for a description of these fields.

Subtyping a Non-Final Entity

If you want to find a new use for an existing entity, you can subtype and rename it. For instance, suppose that you want to track individuals who have already had the role of **IssueOwner**. In this case, it can be useful to create **PastIssueOwner**.

Creating a New Entity

Occasionally, careful review of the base application data model makes it clear that you need to create a new entity. There are many types of base entities within Guidewire applications. However, Guidewire **strongly** recommends in general practice that you always use one of the following types if you create a new entity:

retireable	This type of entity is an extension of the editable entity. It is not possible to delete this entity. It is possible to retire it, however.
versionable	This type of entity has a version and an ID. It is possible to delete entities of this type from the database.

As a general rule, Guidewire recommends the following:

- Make the new entity **versionable** if it is *not* necessary for another entity to refer to the entity through the use of a foreign key.

- Make the entity **retirable** otherwise.

Note: See “Data Objects and the Application Database” on page 193 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.

Note: 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 247 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. You **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.

Note: For information on these two files and localizing search and sort operations in general, see “Localized Search and Sort” on page 525.

Naming Guidelines 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.

This section describes the following naming guidelines:

- Use Allowable Characters Only
- Add a Prefix or Suffix to Avoid Name Conflicts
- Use the Singular Except for Arrays

Use Allowable Characters Only

An extension name *cannot* start with a digit. It must start with an alpha character. Other than that, an extension name can contain letters, numbers, or underscores (_). Guidewire does not permit any other characters in extension names.

Add a Prefix or Suffix to Avoid Name Conflicts

To avoid naming conflicts with base ClaimCenter entities, Guidewire recommends that you add `Ext` as either a prefix or a suffix to your extension names for entities and fields. This ensures that there is no conflict with ClaimCenter entities that Guidewire adds or changes in the future.

- **Using Ext_ as a prefix**—With this approach, start all your extension names with `Ext_`. For example, name an entity that represents the service area covered by a vendor `Ext_ServiceArea`.
- **Using _Ext as a suffix** — With this approach, end all your extension names with `_Ext`. For example, name an entity that represents a person’s credit history `CreditHistory_Ext`.

The primary difference between these approaches is how various development resources, such as in the *Data Dictionary* or the Studio help system, list the extensions.

- If you want to group your extensions together in the list, use the prefix, and ClaimCenter alphabetizes the group together under `Ext`.
- If you want ClaimCenter to alphabetize your extensions in the list by their meaningful names, use the suffix.

IMPORTANT Guidewire **strongly** recommends that you choose one of the extension naming formats and then use it consistently thereafter. In general practice, it is not a good idea to mix naming styles.

Use the Singular Except for Arrays

Guidewire recommends that you name most fields with a singular word such as `Claim` or `Note`. However, as an array field references a list of one or more objects, Guidewire recommends that you name it with a plural word.

For example, `Claim.Description` and `Claim.Policy` are single fields on a `Claim` entity, but `Claim.Notes` is an array of multiple notes. Also, for arrays fields that are extensions, make the primary name plural and not the `Ext` prefix or suffix. For example, use `Ext_MedTreatments` or `MedTreatments_Ext`, and not `MedTreatment_Exts`.

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 the **Data Model Extensions → extensions** folder.
 - b. Select **New → Other** file from the right-click menu.
 - c. Enter the file name, adding the .eti file extension. Studio does not do this for you. Name the file `<entity>.eti`.
2. Within the file, use one of the following XML tags to define the entity.

Data entity	See
<code><component></code>	"Component Data Objects" on page 196
<code><delegate></code>	"Delegate Data Objects" on page 197
<code><entity></code>	"Entity Data Objects" on page 200
<code><extension></code>	"Extension Data Objects" on page 203
<code><nonPersistentEntity></code>	"Nonpersistent Entity Data Objects" on page 204
<code><subtype></code>	"Subtype Data Objects" on page 205
<code><viewEntity></code>	"View Entity Data Objects" on page 207
<code><viewEntityExtension></code>	"View Entity Extension Data Objects" on page 208

3. Add fields to your new data entity. For example:

XML tag	Use to add
<code><array></code>	An array of entities
<code><column></code>	A field with a simple data type
<code><foreignkey></code>	A field referencing another entity
<code><typekey></code>	A field with a typelist

See "Data Object Subelements" on page 209 for information on the possible XML elements that you can add to your new entity definition.

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 254 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.

WARNING 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 .ttx) 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. Within the Studio Resources tree, navigate to the object file for which you want to create an extension. You can do this in several different ways:
 - a. Select CRTL-N and enter the object name in the search field. If multiple versions of this file already exist (.eti and .eix, for example), enter the extension as well to ensure that you find the correct file. Studio opens the Resources tree and highlights the file name.
 - b. Expand all the Data Model Extensions folders and visually search for the data object file.
 - c. Select the Data Model Extensions folder, right-click and use the Find in path... command to search for the file.
 2. Select (highlight) the file, right-click, and select Create extension file from the submenu. Studio creates a basically empty extension file named <entity>.etx, places it in the Data Model Extensions → extensions 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 you do not see the Create extension file right-click command, then search in the extensions folder for an existing file.
3. Populate the extension file with the required XML.
 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 254 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 **Data Model Extensions → metadata** subfolders. Guidewire provides a limited number of override elements for use in **.etx** extension files in the **Data Model Extensions → 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.

IMPORTANT Only add override elements to **.etx** files in the **Data Model Extensions → 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 <code>maintenance_tools</code> command. For more information on the <code>createhistogram</code> attribute on the <code>column</code> element, see “ <code><column></code> ” on page 212.
default	Use to change the default value given to the field (column) during new entity creation.
nullok	Use to make the <code>nullok</code> attribute more restrictive. You can only make it more restrictive, for example, changing it from <code>nullok="true"</code> to <code>nullok="false"</code> .
size	Use to change the size of a column. See “A ‘size’ Example” on page 240.
supportsLinguisticSearch	Use to enable linguistic search on a column. See “Localized Search and Sort” on page 525 for details on working with localized database searches.
triggersValidation	Use to determine if ClaimCenter initiates validation on changes to an array, a foreign key, or a one-to-one entity.

Attribute	Description
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.eli):

```
<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 “The <columnParam> Subelement” on page 214 for a description of the <columnParam> element and the column attributes that you can modify using this element.

A ‘size’ Example

You can change the size of the Name column for a Document entity as follows:

1. Open Guidewire Studio.
2. Navigate to configuration → metadata → pl and right-click Document.eli, and then choose 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.eli.

```
<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.etc file.

a. Find the Account.etc file in the Studio Resources tree. You can use CTRL-N to find the file.

b. Select the file, right-click and select **Create extension file** from the submenu.

Studio creates an Account.etc file and places it in the **Data Model Extensions → extensions** 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 237, 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 **Data Model Extensions** → **extensions**.
2. Right-click and select **New** → **Other** file from the submenu.
3. Enter the file name, using the name of the delegate and adding the **.eti** extension. (Studio does not do this for you.) 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.

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.

To implement delegate functionality through a Gosu enhancement

1. Within Studio, navigate to **Classes** → **gw**.

This is an example package. In actual practice, you can place the enhancement in a location that makes business sense.

2. Right-click and select **New → Enhancement** from the submenu. This action opens the **New Enhancement** dialog.
3. Enter the name of the delegate that you created in “Step 1: Create the Delegate Object” on page 242 in the **Type to Enhance** field. Studio automatically generates the enhancement name from the delegate name. After you enter the delegate name, click **OK** to close the dialog and create the enhancement.
4. Enter code in the enhancement to provide the necessary functionality. The delegate automatically has access to all fields and members that you define in the Gosu enhancement.

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 **Data Model Extensions → extensions**.
 2. Right-click and select **New → Other** file from the submenu.
 3. Enter the name of the entity file. You must add the `.eti` extension. Studio does not do this for you. This action creates an empty file. You use this file to associate the delegate with your entity. If necessary, find an existing entity file and use it as a model for the syntax.
- Note:** Guidewire recommends that you add either the Ext prefix or suffix to all entities that you create or extend. If you do so, do so consistently. This means you always use prefixes or that you always use suffixes.
4. Enter the necessary text in this file, using the `<implementsEntity>` element to specify the delegate. For example (in the ClaimCenter base configuration), Guidewire defines the `Claim` entity—in `Claim.eti`—so that it implements a number of delegates, including the `Assignable` and `Validatable` delegates. The definition looks like this:

```
<entity xmlns="http://guidewire.com/datamodel" ... entity="Claim" ... />
<implementsEntity name="Validatable"/>
<implementsEntity name="Assignable"/>
...
```

Step 4: Deploy your Data Model Changes

After completing these steps, you need to deploy your data model changes. If necessary, see “Deploying Data Model Changes to the Application Server” on page 254 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 197 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.

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.

To extend delegate object

1. Navigate to Data Model Extensions → extensions.
2. Right-click and select New → Other file.
3. Enter the name of the delegate that you want to extend and add the .etx extension. (Studio does not do this for you.) 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 241.

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.etc</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> .
<code>Check.eti</code>	<code>metadata/cc</code>	Creates the <code>Check</code> object, which implements <code>EFTDataDelegate</code> .
<code>Contact.etc</code>	<code>extensions</code>	Extends the <code>Contact</code> entity and adds an array of <code>EFTData</code> 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.etc`.

```
<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 187
- “Delegate Data Objects” on page 197
- “`<implementsEntity>`” on page 224
- “Creating a New Delegate Object” on page 241

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 205.

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 209.

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 → Other file** 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>
```

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 189
defining a new entity	"Defining a New Data Entity" on page 237
defining an array field	"<array>" on page 210
defining a foreign key field	"<foreignkey>" on page 220

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 “View Entity Data Objects” on page 207 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 this file.

```
<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 **Create extension file** from the right-click submenu. This opens a mostly blank file. Enter the following in this file.

```
<viewEntityExtension entityName="ActivityDesktopView">
    <viewEntityColumn name="validExt" path="validExt"/>
</viewEntityExtension>
```

These data model changes add a `validExt` column (field) to the `Activity` object, which is also accessible from the `ActivityDesktopView` entity.

Note: The path attribute is always relative to the primary entity on which you base the view.

Removing Objects from the Base Configuration Data Model

It is possible to safely remove certain objects from the base configuration data model. You can only do this, though, 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 entity	extensions	.eti	" Removing a Base Extension Entity " on page 250
Base configuration extension	extensions	.etx	" Removing an Extension to a Base Object " on page 251

Guidewire recommends that you review the material in "Implications of Modifying the Data Model" on page 251 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 **Data Model Extensions → extensions** 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 hiding—or negating—the effects of the extension entity in the data model.

Note: See "Delete Entity Data Objects" on page 200 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 **Data Model Extensions → extensions** 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 attributes 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 a SQL statement in order to synchronize the database with the data model. How complex this SQL is 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 a 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 250
 - “Removing an Extension to a Base Object” on page 251
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 a 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 Guidewire *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 Guidewire *Data Dictionary* so that it reflects your data model changes:

```
gwcc regen-dictionary
```

2. Create a `.war` or `.ear` file using one the following `build` commands:

```
gwcc build-war  
gwcc build-weblogic-ear  
gwcc build-websphere-ear
```

See the “Key ClaimCenter gwcc Commands” on page 43 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. See the *ClaimCenter Installation Guide* for details.

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 255
- “Subtype Mapping Associative Arrays” on page 256
- “Typelist Mapping Associative Arrays” on page 258

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. For example, one type of associative array uses a typekey (a string) as the array index:

State typekey index	Maps to...
Capital["Alaska"]	"Juneau"
Capital["Alabama"]	"Montgomery"
Capital["Arkansas"]	"Little Rock"

There are two necessary parts to an associative array in Gosu:

- Exposing the key set to the type system

- Calculating the value from the key

Associative Array Types

ClaimCenter uses the following types of associative arrays:

Association type	<array> subelement	Description
Subtype mapping	<link-association>	Link associations return at most one element and the return type is an object of the type of the array.
TypeList mapping	<array-association>	Array associations return an array of results that match the typekey. Array associations are always zero, one, or more.

Both association types can generate a `get`, `set`, or a `has` method.

- The `get` method (the getter) fetches the value or values.
- The `set` method (the setter) sets the values by modifying the field on the object.
- The `has` method queries the array to see if it contains an object that matches the criteria.

For example, in the ClaimCenter `Claim` definition file ([Data Model Extensions → metadata → Claim.eti](#)), you see the following XML (simplified for clarity):

```
<entity xmlns="http://guidewire.com/datamodel"
    entity="Claim"
    table="claim"
    type="retireable">
    ...
<array arrayentity="ClaimMetric"
    desc="Metrics related to this claim."
    exportable="false"
    ignoreforevents="true"
    name="ClaimMetrics"
    triggersValidation="false">
    <link-association>
        <subtype-map/>
    </link-association>
    <array-association>
        <typelist-map field="ClaimMetricCategory"/>
    </array-association>
</array>
...
</entity>
```

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 ([Data Model Extensions → metadata → Claim.eti](#)), you see the following XML (simplified for clarity):

```
<entity xmlns="http://guidewire.com/datamodel"
    entity="Claim"
    table="claim"
    type="retireable">
    ...
<array arrayentity="ClaimMetric"
    desc="Metrics related to this claim."
    exportable="false"
    ignoreforevents="true"
    name="ClaimMetrics"
    triggersValidation="false">
    <link-association>
        <subtype-map/>
    </link-association>
```

```
</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, 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*.

The following example code uses the sample data in the Guidewire ClaimCenter base configuration. It first retrieves a specific claim object and then uses that object as the base entity from which to retrieve array member properties.

```
var clm = find(c in Claim where c.ClaimNumber == "235-53-365870")
var thisClaim = clm.getAtMostOneRow()

print("AllEscalatedActivitiesClaimMetric\tClaim Metric Category = "
    + thisClaim.AllEscalatedActivitiesClaimMetric.ClaimMetricCategory.DisplayName)
print("AllEscalatedActivitiesClaimMetric\tDisplay Value = "
    + thisClaim.AllEscalatedActivitiesClaimMetric.DisplayValue)
print("AllEscalatedActivitiesClaimMetric\tReach Yellow Time = "
    + thisClaim.AllEscalatedActivitiesClaimMetric.ReachYellowTime)
```

The output of running this code in the Gosu tester:

```
AllEscalatedActivitiesClaimMetric      Claim Metric Category = Claim Activity
AllEscalatedActivitiesClaimMetric      Display Value = 0
AllEscalatedActivitiesClaimMetric      Reach Yellow Time = null
```

Setting Array Member Values

There are several issues with setting associative array member values, including:

- You can use a `find` query to retrieve a specific entity instance. However, the result of the `find` query is read-only. You must add the retrieved entity to a bundle to be able to manipulate its fields.
- 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*.

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 query = find(c in Claim where c.ClaimNumber == "235-53-365870")
var thisClaim = query.getAtMostOneRow()

//Result of find query is read-only, need to get current bundle and add entity to bundle
var bundle = Transaction.getCurrent()
bundle.add(thisClaim)

print("AllEscalatedActivitiesClaimMetric\tReach Yellow Time = "
    + thisClaim.AllEscalatedActivitiesClaimMetric.ReachYellowTime)
thisClaim.AllEscalatedActivitiesClaimMetric.ReachYellowTime = todaysDate

print("\nAfter modifying the ReachYellowTime value...\n")
print("AllEscalatedActivitiesClaimMetric\tReach Yellow Time = "
    + thisClaim.AllEscalatedActivitiesClaimMetric.ReachYellowTime)
```

The output of running this code in the Gosu tester:

```
AllEscalatedActivitiesClaimMetricReach Yellow Time = null
    After modifying the ReachYellowTime value...
AllEscalatedActivitiesClaimMetricReach Yellow Time = 2010-05-21
```

TypeList Mapping Associative Arrays

You use a typelist map to partition array objects based on a typelist field (typecode) in the `<array>` element. In the ClaimCenter base configuration, the `ClaimMetrics` array on `Claim` contains a typelist mapping and the previously described subtype mapping.

```
<entity xmlns="http://guidewire.com/datamodel"
    entity="Claim"
    table="claim"
    type="retireable">
    ...
    <array arrayentity="ClaimMetric"
        desc="Metrics related to this claim."
        exportable="false"
        ignoreforevents="true"
        name="ClaimMetrics"
        triggersValidation="false">
        <array-association>
            ...
            <typelist-map field="ClaimMetricCategory"/>
        </array-association>
    </array>
    ...
</entity>
```

The `<typelist-map>` element requires that you set a value for the `field` attribute. This attribute specifies the typelist to use to partition the array.

IMPORTANT *It is an error to specify a typelist mapping on a field that is not a typekey.*

Associative arrays of type `<array-association>` are different from those created using `<link-association>` in that they can return more than a single element. In this case, the code creates an array of `ClaimMetric` objects named `ClaimMetrics`. Each `ClaimMetric` object, and all subtype objects of it, contain a property called `ClaimMetricCategory`. The array definition code utilizes that fact and uses the `ClaimMetricCategory` typelist as a partitioning agent.

The `ClaimMetricCategory` typelist contains three typecodes, which are:

- `ClaimActivityMetrics`
- `ClaimFinancialMetrics`
- `OverallClaimMetrics`

Each typecode specifies a category, which contains multiple `ClaimMetric` object subtypes. For example, the `OverallClaimMetrics` category contains two `ClaimMetric` subtypes:

- `DaysInitialContactWithInsuredClaimMetric`
- `DaysOpenClaimMetric`

In another example from the ClaimCenter base configuration, you see the following defined for `ReserveLine`.

```
<entity entity="ReserveLine"
        xmlns="http://guidewire.com/datamodel"
        ...
        table="reserveline"
        type="retireable">
    ...
    <array arrayentity="TAccount"
          arrayfield="ReserveLine"
          name="TAccounts"
          ...
          <link-association hasGetter="true" hasSetter="true">
              <typelist-map field="TAccountType"/>
          </link-association>
      </array>
    ...
</entity>
```

In this case, the array definition code creates a `<link-association>` array of `TAccount` objects and partitions the array by the `TAccountType` typelist typecodes.

Working with Array Values Using Typelist Mapping

To retrieve an array value, 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): <ul style="list-style-type: none">• <code>ReachRedTime</code>• <code>ReachYellowTime</code>• <code>Skipped</code>

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.)

```
var clm = find(c in Claim where c.ClaimNumber == "235-53-365870")
var thisClaim = clm.getAtMostOneRow()

for (time in thisClaim.OverallClaimMetrics) {
    print(time.Subtype.DisplayName + ": ReachYellowTime = " + time.ReachYellowTime)
}
```

The output of running this code in the Gosu tester (for example):

```
Initial Contact with Insured (Days): ReachYellowTime = 2010-02-01
Days Open: ReachYellowTime = 2010-05-10
```

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 = find(c in Claim where c.ClaimNumber == "235-53-365870")
var thisClaim = clm.getAtMostOneRow()
var thisReserveLine = thisClaim.ReserveLines.first()

print(thisReserveLine)
print(thisReserveLine.cashout.CreateTime)
```

The output of running this code in the Gosu tester (for example):

```
(3) 3rd Party Vehicle - Bo Simpson; Claim Cost/Auto body
Wed Feb 10 16:42:14 PST 2010
```

Setting Array Member Values

The following example code also uses the sample data in the Guidewire ClaimCenter base configuration. It uses a `find` query to retrieve a specific claim entity. As the result of the `find` 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

var todaysDate = java.util.Date.getCurrentDate
var query = find(c in Claim where c.ClaimNumber == "235-53-365870")
var thisClaim = query.getAtMostOneRow()

//Result of find query is read-only, need to get current bundle and add entity to bundle
var bundle = Transaction.getCurrent()
bundle.add(thisClaim)

//Print out the current values for the ClaimMetric.ReachYellowTime field on each subtype
for ("Subtype - " + color in thisClaim.OverallClaimMetrics) {
    print(color.Subtype.DisplayName + ": \n\tReachYellowColor = " + 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 + ": \n\tReachYellowColor = " +
        color.ReachYellowTime)
}
```

The output of running this code in the Gosu tester:

```
Subtype - Initial Contact with Insured (Days): ReachYellowColor = 2010-02-01  
Subtype - Days Open: ReachYellowColor = 2010-08-13
```

After modifying the values...

```
Subtype - Initial Contact with Insured (Days): ReachYellowColor = 2010-02-22  
Subtype - Days Open: ReachYellowColor = 2010-02-22
```


Example: Creating Assignable Entities

This topic describes how to modify the Guidewire data model. It illustrates how to construct a data entity and make it assignable.

This topic includes:

- “Creating an Assignable Extension Entity” on page 263
- “Making Your Extension Entity Eligible for Round-Robin Assignment” on page 267
- “Creating an Assignable Extension Entity that Uses Foreign Keys” on page 271

Creating an Assignable Extension Entity

To create an assignable extension entity, perform the following steps:

- Step 1: Create Extension Entity UserAssignableEntityExt
- Step 2: Create Assignment Class NewAssignableMethodsImpl
- Step 3: Test Assign Your Extension Entity

IMPORTANT The code and objects in this example refer to Guidewire ClaimCenter. However, the principals involved in extending the data model apply to all Guidewire applications.

Step 1: Create Extension Entity UserAssignableEntityExt

The first step in creating an assignable entity is to create the entity (or extend the entity if it already exists) and have that entity implement the following:

- Assignable delegate class
- CCAssignable delegate class

- CCAssignableMethods interface

The entity must link to a Gosu class that defines the necessary assignment methods and properties. In the first step, you defined NewAssignableMethodsImpl, which implemented the CCAssignableMethodsBaseImpl super-class. The following procedure creates a new extension entity that implements this delegate class.

To create entity UserAssignableEntityExt

1. Create the new extension entity file.

- a. In Guidewire Studio, navigate to **Data Model Extensions** → **extensions**, right-click and select **Other file**.
- b. Enter the name of the extension entity in the text field. You must also add the **.eti** extension (if you are creating a new entity) or the **.etx** extension (if you are extending an existing entity). Studio does not add the extension for you.

2. Populate this file with the following code.

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
    desc="CCAssignable extension entity, for testing."
    entity="UserAssignableEntityExt"
    platform="false"
    table="userassignext"
    type="retireable">
    <fulldescription>
        Simple entity that you can assign. This entity does not have a foreign key.
    </fulldescription>
    <implementsEntity name="Assignable"/>
    <implementsEntity name="CCAssignable"/>
    <implementsInterface iface="gw.api.assignment.CCAssignableMethods"
        impl="gw.assignment.NewAssignableMethodsImpl"/>
    <column name="TestColumn" type="varchar" nullok="true" desc="varchar extension">
        <columnParam name="size" value="60"/>
    </column>
    <column name="ExtEntityUpdateTime" type="datetime"/>
    <column name="NewlyAssignedActivities" type="integer"/>
    <column name="IntegerExt"
        type="integer"
        default="12"
        nullok="true"
        desc="integer extension; default value of '12'" />
    <typekey name="LossType"
        typelist="LossType"
        nullok="true"
        desc="High level claim type (for example, Auto or Property)." />
    <typekey name="PolicyType"
        typelist="PolicyType"
        nullok="true"
        desc="High level policy type (for example, Auto or Property)." />
    <index name="internal1">
        <indexcol name="TestColumn" keyposition="1"/>
    </index>
</entity>
```

Notice the following:

- Entity UserAssignableEntityExt implements the **Assignable** and **CCAssignable** delegates.
- Entity UserAssignableEntityExt uses class **NewAssignableMethodsImpl**, which implements the **CCAssignableMethods** interface, to define the necessary assignment methods and properties. (This delegate class does not exist yet. However, you create it in the next step.)

See Also

- For information on how to extend the data model, see “[Modifying the Base Data Model](#)” on page 233.
- For information on the data model in general, see “[The ClaimCenter Data Model](#)” on page 187.

Step 2: Create Assignment Class NewAssignableMethodsImpl

Any new assignable entity must implement the **CCAssignableMethods** delegate. The following procedure defines a delegate class that provides an implementation of the methods in the **CCAssignableMethods** interface.

In this case, the `NewAssignableMethodsImpl` class extends the abstract superclass `CCAssignableMethodsBaseImpl`, which provides standard implementations of some of this functionality.

To create class `NewAssignableMethodsImpl`

1. Create a new class file (in package `gw.assignment`) and name it `NewAssignableMethodsImpl`.
 - a. In Studio, navigate to **Classes** → **gw** → **assignment**.
 - b. Right-click and select **New** → **Class**.
 - c. Enter the name of the class in the text box. (You do not need to enter the file extension. Studio does this for you.) For this example, enter the following:

`NewAssignableMethodsImpl`

Studio creates a skeleton class file in the `assignment` folder and opens the file in a view tab.

2. Have your class implement one of the following:

- Interface `gw.api.assignment.CCAssignableMethods`
- Abstract superclass `gw.api.assignment.CCAssignableMethodsBaseImpl`

Which one you choose depends on your business needs. In either case, you must provide method bodies for all unimplemented methods. To determine which methods you need to implement, you need merely have your class implement the interface (or extend the abstract class).

In other words, enter the following:

```
class NewAssignableMethodsImpl implements gw.api.assignment.CCAssignableMethods
```

Studio marks this line of code as invalid. This is because you must provide implementations of all the methods that the interface defines, which you have not yet done. If you place your cursor in the interface name, Studio marks the interface name in red. Studio underlines this line of code and displays a tooltip asking if you want to insert the missing methods. Accept this help. Studio then inserts skeleton code bodies for all the required methods.

3. In this example, enter the following code. This makes the `NewAssignableMethodsImpl` class extend abstract superclass `CCAssignableMethodsBaseImpl`.

```
package gw.assignment
uses gw.api.assignment.CCAssignableMethods
uses gw.api.database.IQueryResult
uses gw.api.assignment.CCAssignableMethodsBaseImpl
uses gw.entity.ILinkPropertyInfo

class NewAssignableMethodsImpl extends CCAssignableMethodsBaseImpl {
    construct(testEntity : UserAssignableEntityExt) {
        super(testEntity) // added for example
    }

    override property get Owner() : UserAssignableEntityExt {
        return super.Owner as UserAssignableEntityExt
    }

    override function shouldCascadeAssignment(parent : Assignable, defaultOwnerUserId : Key,
        defaultGroupId : Key) : boolean {
        return false // added for example
    }

    override property get ChildrenForCascadeAssignment() : List<CCAssignableMethods> {
        return {} // added for example
    }

    override property get OwningClaim() : Claim {
        return null //## todo: Implement me
    }

    override property get AssignmentReviewActivityLinkProperty() : ILinkPropertyInfo {
        return null //## todo: Implement me
    }

    override property get AssignmentReviewActivitySubject() : String {
        return null //## todo: Implement me
    }
}
```

```
}

override function pushAssignmentPopup(p0 : List<CCAssignableMethods>) {
    //## todo: Implement me
}

}
```

Step 3: Test Assign Your Extension Entity

After you create an assignable entity, it is important to test your work to verify that you have performed the previous steps correctly.

To test assignable entity UserAssignableEntityExt

1. Stop and restart Studio. Although not strictly required, it is a useful test to determine if you have inadvertently made a typing mistake, for example.
2. Start (or restart) the application server.

IMPORTANT Any time you change the base data model, you must restart the application server to force a database upgrade.

3. Correct any problems that occur. If you are starting the application server, ensure that there is at least some demonstration data available for the next step. This is especially true for the user name that you list in the query statement in the test code.

4. Connect Studio to the running application server.

- a. Click the **Disconnected** link in the bottom right-hand corner of the Studio screen.

- b. Clear the Server URL field and enter the following:

`http://localhost:8080/cc`

- c. Click **Log In**. After a few seconds, Studio displays a connection URL in place of the **Disconnected** link. Do not proceed until you see this message.

5. Open the Studio Gosu Tester by clicking its icon on the Studio toolbar.  (It is the first icon on the right-hand side of the toolbar.)

6. Enter the following code in the **Gosu Source** tab.

```
var _user = find (u in User where exists ( c in User.Credential where c.UserName ==
    "applegate")).getFirstResult()
var _group = _user.GroupUsers[0].Group
var newExtEntity = new UserAssignableEntityExt()

print (newExtEntity.assign(_group, _user)) // Assigns the entity to the specified user and group
print (newExtEntity.AssignedUser)
print (newExtEntity.AssignedGroup)
```

7. Click **Run**. The Gosu Tester **Runtime Output** tab outputs the following:

```
true          // The assign method succeeded, thus it returns true.
Andy Applegate // The user assigned the newExtEntity (UserAssignableEntityExt) object.
Auto1 - TeamA  // The group assigned the newExtEntity (UserAssignableEntityExt) object.
```

Making Your Extension Entity Eligible for Round-Robin Assignment

To assign an entity through round-robin assignment, you invoke the `assignUserByRoundRobin` method on the entity. To implement round-robin assignment on the assignable entity, you need to add extra tracking fields for the round-robin state to the following entities:

- `DynamicAssignmentState`
- `GroupAssignmentState`
- `GroupUserAssignmentState`

To do so, you need add these fields (columns) as an extension to a base configuration entity. Or, if the base entity does not exist, then you need to create an extension entity with the necessary fields.

Finally, you must create a new Gosu class that extends (subclasses) the `AssignmentType` class. You add methods to this subclass to inform the round-robin infrastructure how to find the newly added fields.

The following procedures assumes the existence of an entity type named `UserAssignableEntityExt`. The steps in “Creating an Assignable Extension Entity” on page 263 describe how to create this entity and make assignable.

This example requires the following steps:

- Step 1: Extend the Assignment State Entities
- Step 2: Subclass Class `AssignmentType`
- Step 3: Create `UserAssignableEntityExtEnhancement`
- Step 4: Test Round Robin Assignment

IMPORTANT The code and objects in this example refer to Guidewire ClaimCenter. However, the principals involved in extending the data model apply to all Guidewire applications.

Step 1: Extend the Assignment State Entities

The first step in creating an entity that you can use with round-robin assignment is to extend the following base configuration entities:

- `DynamicAssignmentState`
- `GroupAssignmentState`
- `GroupUserAssignmentState`

You extend these entities to add fields (columns) on the object that ClaimCenter uses to track information for the round robin state.

To extend the `AssignmentState` entities

1. In Guidewire Studio, open `DynamicAssignmentState.etx` for editing:

- a. Press CTRL-N and start entering the entity type name. Studio displays a number of matches if this file already exists.
- b. **Ensure that you choose the correct file.** There are a number of files with the same name but with different file extensions. In this case, you want the `.etx` extension because an extension file for the entity already exists in the base configuration.

2. In file `DynamicAssignmentState.etx`, enter the following:

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="DynamicAssignmentState">
  <foreignkey columnName="LastTEAEUserID" name="LastTEAEUser" fkentity="User"/>
  <foreignkey columnName="LastTEAEGrpID" name="LastTEAEGrp" fkentity="Group"/>
  <foreignkey columnName="LastTCAUUserID" name="LastTCAUUser" fkentity="User"/>
  <foreignkey columnName="LastTCAGrpID" name="LastTCAGrp" fkentity="Group"/>
```

```
<index desc="Index to help the performance of assign by proximity search"
       name="dynassign1"
       unique="true">
  <indexcol keyposition="1" name="Fingerprint"/>
  <indexcol keyposition="2" name="GroupOnly"/>
  <indexcol keyposition="3" name="Retired"/>
</index>
</extension>
```

3. Repeat this process with file `GroupAssignmentState.etx`. If this file does not exist, then you need to create it. (Right-click the `extensions` folder and select `Other file`. You must give it the `.etx` file extension.)

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="GroupAssignmentState">
  <foreignkey columnName="LastTEAUserID" name="LastTEAUser" fkentity="User"/>
  <foreignkey columnName="LastTEAEGrpID" name="LastTEAEGrp" fkentity="Group"/>
  <column name="TEAELoad" type="integer"/>
  <foreignkey columnName="LastTCAUerID" name="LastTCAUer" fkentity="User"/>
  <foreignkey columnName="LastTCAGrpID" name="LastTCAGrp" fkentity="Group"/>
  <column name="TCALoad" type="integer"/>
</extension>
```

4. Repeat this process with file `GroupUserAssignmentState.etx`. Again, create an `.etx` file if necessary.

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="GroupUserAssignmentState">
  <column name="TEAELoad" type="integer"/>
  <column name="TCALoad" type="integer"/>
</extension>
```

Step 2: Subclass Class AssignmentType

Next, you need to create a subclass of the `AssignmentType` class. This class contains getter and setter methods that inform the round-robin mechanism of the location of the entity fields that you created in the previous step.

To subclass class AssignmentType

1. Create a new class file and name it `UserAssignableEntityExtAssignmentType`:

- a. In Guidewire Studio, navigate to `Classes` → `gw` → `assignment`.
- b. Right-click and select `New` → `Class` from the menu.
- c. Enter `UserAssignableEntityExtAssignmentType` for the class name. You do not need to add the class `.gs` extension. Studio creates a new skeleton class in the `gw.assignment` package and opens it in a view tab.

2. Enter the following in your class file. Studio indicates that the code is invalid (in the `get AssignableClass` method definition) as you have not yet created the necessary `UserAssignableEntityExt` Gosu enhancement. You create the needed entity enhancement in the next step.

```
package gw.assignment

uses gw.api.assignment.AssignmentType
uses gw.entity.IEntityType

class UserAssignableEntityExtAssignmentType extends AssignmentType {

  construct() {}

  override property get AssignableClass() : IEntityType {
    return UserAssignableEntityExt
  }

  override function getGroupLoad(groupState: GroupAssignmentState) : int {
    return groupState.TEAEload
  }

  override function getLastAssignedGroup(dynamicState: DynamicAssignmentState) : Key {
    return dynamicState.LastTEAEGrp as Key
  }

  override function getLastAssignedGroup(groupState: GroupAssignmentState) : Key {
    return groupState.LastTEAEGrp as Key
  }

  override function getLastAssignedUser(dynamicState: DynamicAssignmentState) : Key {
```

```
        return dynamicState.LastTEAEUser as Key
    }

    override function getLastAssignedUser(groupState: GroupAssignmentState) : Key {
        return groupState.LastTEAEUser as Key
    }

    override function getUserLoad(groupUserState: GroupUserAssignmentState) : int {
        return groupUserState.TEAELoad
    }

    override function setGroupLoad(groupState: GroupAssignmentState, load: int) {
        groupState.TEAEload = load
    }

    override function setLastAssignedGroup(dynamicState: DynamicAssignmentState, group: Key) {
        dynamicState.LastTEAEGrp = dynamicState.Bundle.loadByKey(group) as Group
    }

    override function setLastAssignedGroup(groupState: GroupAssignmentState, group: Key) {
        groupState.LastTEAEGrp = groupState.Bundle.loadByKey(group) as Group
    }

    override function setLastAssignedUser(dynamicState: DynamicAssignmentState, user: Key) {
        dynamicState.LastTEAEUser = dynamicState.Bundle.loadByKey(user) as User
    }

    override function setLastAssignedUser(groupState: GroupAssignmentState, user: Key) {
        groupState.LastTEAEUser = groupState.Bundle.loadByKey(user) as User
    }

    override function setUserLoad(groupUserState: GroupUserAssignmentState, load: int) {
        groupUserState.TEAEload = load
    }
}
```

Step 3: Create UserAssignableEntityExtEnhancement

You need to create an enhancement to the `UserAssignableEntityExt` entity for use in creating entities of that type.

To create enhancement `UserAssignableEntityExtEnhancement`

1. Create a new enhancement file and name it `UserAssignableEntityExtEnhancement`.
 - a. In Guidewire Studio, navigate to `Classes` → `gw` → `assignment`.
 - b. Right-click and select `New` → `Enhancement` from the menu.
 - c. Enter `UserAssignableEntityExt` in the `Type to Enhance` field. The text you enter filters the entities that you can enhance through this dialog.
 - d. If there are multiple selections in the selection box in the middle of the dialog, ensure that you select `UserAssignableEntityExt`. This is the entity that you want to enhance. Studio automatically enters `UserAssignableEntityExtEnhancement` in the `Enhancement Name` field.
 - e. Agree to this name by clicking `OK`.

2. Enter the following in the enhancement file:

```
package gw.assignment
uses gw.api.assignment.AssignmentType

enhancement UserAssignableEntityExtEnhancement : entity.UserAssignableEntityExt {

    static property get ASSIGNMENT_TYPE() : AssignmentType {
        return new UserAssignableEntityExtAssignmentType()
    }
}
```

Step 4: Test Round Robin Assignment

After you create an assignable entity suitable for use with round-robin assignment, it is important to test your work to verify that you have performed the previous steps correctly.

To test assignable entity UserAssignableEntityExt

1. Stop and restart Studio. Although not strictly required, it is a useful test to determine if you have inadvertently made a typing mistake, for example.
2. Start (or restart) the application server.

IMPORTANT Any time you change the base data model, you must restart the application server to force a database upgrade.

3. Correct any problems that occur. If you are starting the application server, ensure that there is at least some demonstration data available for the next step.
4. Connect Studio to the running application server.
5. Open the Studio Gosu Tester by clicking its icon on the Studio toolbar.  (It is the first icon on the right-hand side of the toolbar.)
6. Enter the following code in the **Gosu Source** tab.

```
uses java.lang.Integer  
  
com.guidewire.pl.system.logging.PLLoggerCategory.ASSIGNMENT.setLevel( org.apache.log4j.Level.DEBUG )  
  
var _user = find (u in User where  
    exists ( c in User.Credential where c.UserName == "applegate")).getFirstResult()  
var _group = _user.GroupUsers[0].Group  
var newExtEntity = new UserAssignableEntityExt()  
  
newExtEntity.assign(_group, _user) // Uses the assign method to assign the object to a user and group  
  
print (newExtEntity.AssignedGroup) // Group assigned newExtEntity (UserAssignableEntityExt) object.  
print (newExtEntity.AssignedUser) // User assigned newExtEntity (UserAssignableEntityExt) object.  
print("")  
  
var i : Integer = 0  
var ctr = i + 1  
while (i < 3) {  
    print("Pass " + ctr + ":")  
    print (" " + newExtEntity.assignUserByRoundRobin(false, _group))  
    print (" " + newExtEntity.AssignedUser)  
    i++  
}
```

7. Click **Run**. The Gosu Tester **Runtime Output** tab outputs the following:

```
Auto1 - TeamA  
Andy Applegate // The user assigned the newExtEntity (UserAssignableEntityExt) object.  
  
Pass 1:  
    true // Method assignUserByRoundRobin returns true if it succeeds  
    Andy Applegate // Method assignUserByRoundRobin assigns the object to the next person in the group  
Pass 2:  
    true  
    Sue Smith  
Pass 3:  
    true  
    Betty Baker
```

Creating an Assignable Extension Entity that Uses Foreign Keys

To create an assignable extension entity with foreign key links to `Claim` and `Activity`, perform the following steps:

- Step 1: Create Extension Entity `TestClaimAssignable`
- Step 2: Add Foreign Keys to Assignable Entities
- Step 3: Create New Assignment Type for New Extension Entity
- Step 4: Create Enhancement `TestClaimAssignableEnhancement`
- Step 5: Create Test Class `TestClaimAssignableMethodsImpl`
- Step 6: Add Corresponding Permission for the Extension Entity
- Step 7: Test Assignment of the Assignable Entity

IMPORTANT The code and objects in this example refer to Guidewire ClaimCenter. However, the principals involved in extending the data model apply to all Guidewire applications.

Step 1: Create Extension Entity `TestClaimAssignable`

The first step is to create the assignable entity. As with any assignable entity, it must implement the following:

- `Assignable` delegate class
- `CCAssignable` delegate class
- `CCAssignableMethods` interface

To create entity `TestClaimAssignable`

1. Create a new extension entity file in the `extensions` folder and name it `TestClaimAssignable.eti`.
2. Populate this file with the following code.

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
        desc="CCAssignable extension entity which has a foreign key back to claim, used for testing."
        entity="TestClaimAssignable"
        exportable="true"
        platform="false"
        table="testclaimassign"
        type="retireable">
    <fulldescription>
        Simple test assignable with a foreign key back to claim.
        This, together with the CCAssignableMethods delegate in
        TestClaimAssignableMethodsImpl, allows it to participate in manual assignment,
        assignment to claim owner and cascading.
        It also creates history events as it is assigned.
    </fulldescription>
    <implementsEntity name="Extractable"/>
    <implementsEntity name="Assignable"/>
    <implementsEntity name="CCAssignable"/>
    <implementsInterface iface="gw.api.assignment.CCAssignableMethods"
                         impl="gw.assignment.TestClaimAssignableMethodsImpl"/>
    <foreignkey columnName="ClaimID"
                desc="The Claim for this test assignable."
                exportable="false"
                fkentity="Claim"
                name="Claim"
                nullok="false"/>
</entity>
```

Notice the following:

- Entity `TestClaimAssignable` implements the `Assignable` and `CCAssignable` delegates.

- Entity `TestClaimAssignable` uses class `TestClaimAssignableMethodsImpl`, which implements the `CCAssignableMethods` interface, to define the necessary assignment methods and properties.
- Entity `TestClaimAssignable` has a foreign key back to `Claim`.

Step 2: Add Foreign Keys to Assignable Entities

If you want to link your assignable entity to one of the base configuration assignable entities, you need a foreign key from that base configuration entity to your assignable entity.

To add foreign keys to `TestClaimAssignable`

- Open `Claim.etc` for editing and add the following at an appropriate place in the XML:

```
<array arrayentity="TestClaimAssignable"
       desc="Test assignables for this claim."
       exportable="true"
       name="TestClaimAssignables"/>
```

- Open `Activity.etc` for editing and add the following at an appropriate place in the XML:

```
<foreignkey columnName="TestClaimAssignableID"
            name="TestClaimAssignable"
            fkentity="TestClaimAssignable"/>
```

Step 3: Create New Assignment Type for New Extension Entity

Next, you need to create a subclass of the `AssignmentType` class. This class contains getter and setter methods that inform the round-robin mechanism of the location of the entity fields added to the assignment state entities.

To subclass class `AssignmentType`

- Create a new class file in package `gw.assignment` and name it `TestClaimAssignableAssignmentType.gs`.
- Enter the following in your class file.

```
package gw.assignment

uses gw.api.assignment.AssignmentType
uses gw.entity.IEntityType

class TestClaimAssignableAssignmentType extends AssignmentType {
    construct() {}

    override property getAssignableClass() : IEntityType {
        return TestClaimAssignable
    }

    override function getGroupLoad(groupState: GroupAssignmentState) : int {
        return groupState.TCALoad
    }

    override function getLastAssignedGroup(dynamicState: DynamicAssignmentState) : Key {
        return dynamicState.LastTCAGrp as Key
    }

    override function getLastAssignedGroup(groupState: GroupAssignmentState) : Key {
        return groupState.LastTCAGrp as Key
    }

    override function getLastAssignedUser(dynamicState: DynamicAssignmentState) : Key {
        return dynamicState.LastTCAUser as Key
    }

    override function getLastAssignedUser(groupState: GroupAssignmentState) : Key {
        return groupState.LastTCAUser as Key
    }

    override function getUserLoad(groupUserState: GroupUserAssignmentState) : int {
        return groupUserState.TCALoad
    }

    override function setGroupLoad(groupState: GroupAssignmentState, load: int) {
```

```
        groupState.TCALoad = load
    }

    override function setLastAssignedGroup(dynamicState: DynamicAssignmentState, group: Key) {
        dynamicState.LastTCAGrp = dynamicState.Bundle.loadByKey(group) as Group
    }

    override function setLastAssignedGroup(groupState: GroupAssignmentState, group: Key) {
        groupState.LastTCAGrp = groupState.Bundle.loadByKey(group) as Group
    }

    override function setLastAssignedUser(dynamicState: DynamicAssignmentState, user: Key) {
        dynamicState.LastTCAUser = dynamicState.Bundle.loadByKey(user) as User
    }

    override function setLastAssignedUser(groupState: GroupAssignmentState, user: Key) {
        groupState.LastTCAUser = groupState.Bundle.loadByKey(user) as User
    }

    override function setUserLoad(groupUserState: GroupUserAssignmentState, load: int) {
        groupUserState.TCALoad = load
    }

}
```

Step 4: Create Enhancement TestClaimAssignableEnhancement

Create an enhancement to the `TestClaimAssignable` entity for use in creating entities of that type.

To create enhancement `TestClaimAssignableEnhancement`

1. Create a new enhancement file in package `gw.assignment` and name it `TestClaimAssignableEnhancement`.
 - a. Select **New → Enhancement** from the right-click contextual menu on package `gw.assignment`.
 - b. Enter `TestClaim` in the **Type to Enhance** field and select `TestClaimAssignable` from the list of types.
2. Enter the following in the enhancement file:

```
package gw.assignment

uses gw.api.assignment.AssignmentType

enhancement TestClaimAssignableEnhancement : TestClaimAssignable {

    static property get ASSIGNMENT_TYPE() : AssignmentType {
        return new TestClaimAssignableAssignmentType();
    }

}
```

Step 5: Create Test Class TestClaimAssignableMethodsImpl

Previously, you created the `TestClaimAssignable` entity type and defined it (in part) using the following:

```
<implementsInterface iface="gw.api.assignment.CCAssignableMethods"
    impl="gw.assignment.TestClaimAssignableMethodsImpl"/>
```

This code defines a delegate class that provides the implementation of the methods in the `CCAssignableMethods` interface. In this case, the `NewAssignableMethodsImpl` class extends the abstract superclass `CCAssignableMethodsBaseImpl`, which provides standard implementations of some of this functionality.

To create class `TestClaimAssignableMethodsImpl`

1. Create a new class file in package `gw.assignment` and name it `TestClaimAssignableMethodsImpl`.
2. Enter the following code. This makes the `TestClaimAssignableMethodsImpl` class extend abstract superclass `CCAssignableMethodsBaseImpl`.

```
package gw.assignment

uses gw.api.assignment.CCAssignableMethodsBaseImpl
uses gw.entity.ILinkPropertyInfo
uses gw.api.assignment.CCAssignableMethods
```

```


/**
 * Example CCAssignableMethodsImplementation for an assignable entity that is related to a claim,
 * and which can be manually assigned, assigned to claim owner or cascaded. Also creates a history
 * event as it is assigned.
 */

class TestClaimAssignableMethodsImpl extends CCAssignableMethodsBaseImpl {

    construct(testEntity : TestClaimAssignable) {
        super(testEntity)
    }

    override property get Owner() : TestClaimAssignable {
        return super.Owner as TestClaimAssignable
    }

    override property get OwningClaim() : Claim {
        return Owner.Claim
    }

    override property get AssignmentReviewActivityLinkProperty() : ILinkPropertyInfo {
        return Activity.Type.TypeInfo.getProperty("TestClaimAssignable") as ILinkPropertyInfo
    }

    override property get AssignmentReviewActivitySubject() : String {
        return "Test Claim Assignable Assignment Review"
    }

    override function pushAssignmentPopup(assignables : List<CCAssignableMethods>) {
        // needed for UI
        AssignmentPopupUtil.pushAssignTestClaimAssignables(assignables.whereTypeIs(TestClaimAssignable).
            toTypedArray())
    }

    override function createAssignmentHistoryEvent(assignable : Assignable) : History {
        var result = super.createAssignmentHistoryEvent(assignable)
        result.Description = "Test Claim Assignable History Event"
        return result
    }

    override function shouldCascadeAssignment(parent : Assignable,
        defaultOwnerId : Key,
        defaultGroupId : Key) : boolean {
        return true
    }

    override property get ChildrenForCascadeAssignment() : List<CCAssignableMethods> {
        return {}
    }
}


```

At this point, Studio indicates that your newly created class contains a serious error as there is no method definition for `AssignmentPopupUtil.pushAssignTestClaimAssignables`. The next step resolves this issue.

3. Open `gw.assignment.AssignmentPopupUtil` for editing and add the following:

```


static function pushAssignTestClaimAssignables(testclaimassignables : TestClaimAssignable[]) {
    // Implementation do do
}


```

As the `pushAssignmentPopup` method on `TestClassAssignableMethodsImpl` calls the `pushAssignTestClaimAssignables` method in `AssignmentPopuUtil`, you need to provide at least a skeleton body of this method.

Step 6: Add Corresponding Permission for the Extension Entity

Each extension entity that has a foreign key to another entity (Claim or Activity, for example) must do the following:

- Define a permission of `permKey="own"` in `security-config.xml`.
- Add this permission to the `SystemPermissionType` typelist.
- Assign this permission to a user role.
- Assign this role to any user that wants to assign the new assignable entity.

If you do not perform these steps correctly, you see the following error if you attempt to assign your assignable entity:

```
com.guidewire.pl.system.exception.InsufficientPermissionException: No user defined  
at com.guidewire.pl.system.security.PermCheck.setAndCheckAssign(PermCheck.java:81)  
at com.guidewire.pl.domain.assignment.implAssignableImpl.assign(AssignableImpl.java:236)  
...
```

To add a permission handler for the extension entity

You use file `security-config.xml` to define various system permissions for the different data model entities. The following code adds a permission handler for the assignable extension entity.

1. Open file `security-config.xml` for editing. You can find it in the **Other Resources** folder.
2. Add the following to this file to create a permission that you can assign to someone to enable that person to assign your entity.

```
<!-- Permission key needed for assigning extension entity in testing-->  
<StaticHandler entity="UserAssignableEntityExt" permKey="own">  
  <SystemPermType code="ownsensclaim"/>  
  <SystemPermType code="InternalTools"/>  
  <SystemPermType code="ext_entity_perms"/>  
</StaticHandler>  
<StaticHandler entity="TestClaimAssignable" permKey="own">  
  <SystemPermType code="ownsensclaim"/>  
  <SystemPermType code="InternalTools"/>  
  <SystemPermType code="ext_entity_perms"/>  
</StaticHandler>
```

To add the permission to the system permission typelist

You need to add the new system permission to the `SystemPermissionType` typelist. (You then associate that permission with the user role allowed to work with the new assignable entity.)

1. First, search for the `SystemPermissionType` typelist using CTRL-N
2. Click **Add** and enter the following:

Field	Entry
Code	ext_entity_perms
Name	Own ext entity
Description	Permission to access extension entity

To assign a permission to a user role

If you want a user to be able to assign your assignable object, then you first need to assign the `ext_entity_perms` system permission to a specific user role. You then need to assign this role to a specific user (if the user does not already have that role).

IMPORTANT To see your new system permission in Guidewire ClaimCenter, stop and restart Studio and the application server.

1. Log into your Guidewire installation under an administrative account. (This example assumes the use of Guidewire ClaimCenter.)
2. Navigate to the **Administration** → **Roles** tab.
3. Assign the `ext_entity_perms` system permission to a specific user role, for example, *Adjuster*.
4. Assign the role with the necessary permission to a specific user such as *aapplegate*.

Note: For information about roles and permissions, see “Security: Roles, Permissions, and Access Controls” in the *ClaimCenter Application Guide*.

Step 7: Test Assignment of the Assignable Entity

After you create your assignable entity, it is important to test your work to verify that you have performed the previous steps correctly.

To test assignable entity TestClaimAssignable

1. Stop and restart Studio. Although not strictly required, it is a useful test to determine if you have inadvertently made a typing mistake, for example.
2. Start (or restart) the application server.

IMPORTANT Any time you change the base data model, you must restart the application server to force a database upgrade.

3. Correct any problems that occur. If you are starting the application server, ensure that there is at least some demonstration data available for the next step.
4. Connect Studio to the running application server.
5. Open the Studio Gosu Tester by clicking its icon on the Studio toolbar.  (It is the first icon on the right-hand side of the toolbar.)
6. Enter the following code in the **Gosu Source** tab.

```
uses gw.transaction.Bundle
uses gw.lang.reflect.TypeSystem
uses java.lang.Integer

//Use the user name of a valid user in your sample data
var _user = find(u in User where exists ( c in User.Credential where c.UserName ==
    "applegate")).getFirstResult()
var _group = _user.GroupUsers[0].Group

//Use a claim number for a valid claim in your sample data
var _claim = find(c in Claim where c.ClaimNumber == "235-53-365870").AtMostOneRow
print ("Claim number = " + _claim.ClaimNumber)
print("")

print("Testing assignable entity linked to Claim")
var testAssignable = new TestClaimAssignable()
testAssignable["Claim"] = _claim
testAssignable.assign(_group, _user)
print("Assigning claim (with claim number = " + testAssignable.Claim +
    ") to the test assignment object ")
print (" The assigned user is " + testAssignable.AssignedUser)
print (" The assigned group is " + testAssignable.AssignedGroup)
print("")

print("Testing round-robin assignment")
//Print out the members of the chosen group
print("Chosen group " + _group.DisplayName + " contains the following members:")
_group.Users.each(\ name -> print (" "+ name.User.DisplayName))
print("")

print("Iterating through round-robin assignment of the test assignment object")
var i : Integer = 0
var ctr : Integer

while (i < 5) {
    if (testAssignable.assignUserByRoundRobin(false, _group) != true) {
        print("Cannot perform round-robin assignment.")
        break
    }
    ctr = i + 1
    print (" Pass" + ctr + ": " + "For assignUserByRoundRobin, AssignedUser = " +
        testAssignable.AssignedUser)
    i++
}
```

7. Click Run. The Gosu Tester Runtime Output tab outputs the following:

```
Claim number = 235-53-365870

Testing assignable entity linked to Claim
Assigning claim (with claim number = 235-53-365870) to the test assignment object
The assigned user is Andy Applegate
The assigned group is Auto1 - TeamA

Testing round-robin assignment
Chosen group Auto1 - TeamA contains the following members:
Sue Smith
Andy Applegate
Charles Shaw
Betty Baker
Cathy Clark
Eugene Nyugen
Felicity Wagner
Heidi Johnson
Elizabeth Lee
Dan Henson
Gary Wang
Chris Craft

Iterating through round-robin assignment of the test assignment object
Pass1: For assignUserByRoundRobin, AssignedUser = Andy Applegate
Pass2: For assignUserByRoundRobin, AssignedUser = Betty Baker
Pass3: For assignUserByRoundRobin, AssignedUser = Chris Craft
Pass4: For assignUserByRoundRobin, AssignedUser = Dan Henson
Pass5: For assignUserByRoundRobin, AssignedUser = Eugene Nyugen
```


The Domain Graph

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

This topic includes:

- “What is the Domain Graph?” on page 280
- “Object Ownership and the Domain Graph” on page 281
- “Accessing the Domain Graph” on page 282
- “Adding Objects to the Domain Graph” on page 284
- “Graph Validation Checks” on page 288
- “Working with Changes to the Data Model” on page 289
- “Working with Shared Entity Data” on page 290
- “Working with Cycles” on page 290

See also

- “Archiving” on page 87 in the *Application Guide* – information on archiving claims, searching for archived claims, and restoring archived claims.
- “Archive Parameters” on page 39 in the *Configuration Guide* – discussion on the configuration parameters used in claims archiving.
- “Archiving Claims” on page 665 in the *Configuration Guide* – information on configuring claims archiving, selecting claims for archiving, and archiving and the object (domain) graph.
- “Archiving Integration” on page 285 in the *Integration Guide* – describes the archiving integration flow, storage and retrieval integration, and the `IArchiveSource` plugin interface.
- “Archive” on page 48 in the *Rules Guide* – information on base configuration archive rules and their use in detecting archive events and managing the claims archive and restore process.
- “Logging Successfully Archived Claims” on page 37 in the *System Administration Guide*.
- “Purging Unwanted Claims” on page 58 in the *System Administration Guide*.
- “Archive Info” on page 160 in the *System Administration Guide*.

- “Upgrading Archived Entities” on page 62 in the *Upgrade Guide*.

IMPORTANT To increase performance, most customers find increased hardware more cost effective than archiving unless their volume exceeds one million claims or more. Guidewire strongly recommends that you contact Customer Support before implementing archiving to help your company with this analysis.

What is the Domain Graph?

The domain graph is a *set of entities*. This 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. For example, in Guidewire ClaimCenter, the root entity is the `Claim` 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.

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.

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. *If you attempt to share a graph entity between two or more domain instances, then archiving fails.*

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

IMPORTANT The use of object archiving within your application instance is not automatic. To enable archiving within Guidewire ClaimCenter, you must set configuration parameter `ArchiveEnabled` to `true` (in `config.xml`).

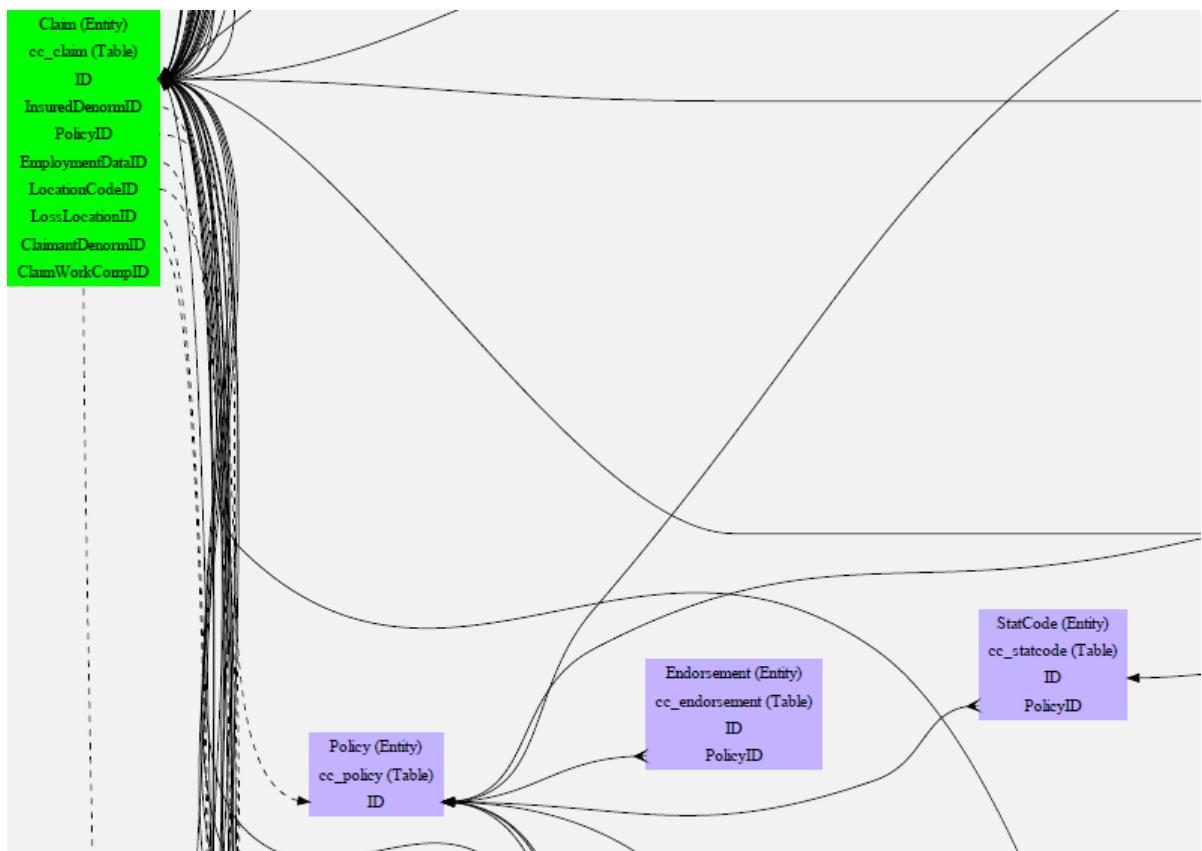
Understanding Graph Notation

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 using human-readable text, but which specialized software can use to generate a visual representation of the graph. DOT graph files generally end with the `.dot` extension.

For example, in ClaimCenter, the domain graph representation in the DOT language begins like this:

```
digraph MyGraph {
    rankdir=RL ranksep=1.0 node [shape=record height=.1];
    Claim[style=filled,color=green,label="<Claim (Entity)>Claim (Entity)|<cc_claim (Table)>cc_claim (Table)|<ID>ID|<InsuredDenormID>InsuredDenormID|<PolicyID>PolicyID|<EmploymentDataID>EmploymentDataID|<LocationCodeID>LocationCodeID|<LossLocationID>LossLocationID|<ClaimantDenormID>ClaimantDenormID|<ClaimWorkCompID>ClaimWorkCompID"];
    ClaimAccess[style=filled,color=".7 .3 1.0",label="<ClaimAccess (Entity)>ClaimAccess (Entity)|<cc_claimaccess (Table)>cc_claimaccess (Table)|<ID>ID|<ClaimID>ClaimID"];
    Incident[style=filled,color=".7 .3 1.0",label="<Incident (Entity)>Incident (Entity)|<cc_incident (Table)>cc_incident (Table)|<
```

The visual representation of the ClaimCenter domain graph looks—in a very small part—similar to the following:



Within the graph, the notation has the following meaning:

- A solid line with an arrow at its head represents a foreign key.
- A solid line with an arrow at its head and a crow's foot at its tail represents an array. The arrow points to the entity that has the array property and the crow's foot represents the table that contains the rows in the array.
- A dashed line with an arrow at its head represents a foreign key in which `owned=true`. The entity at the arrow end of the edge is the owned entity.
- An arrow in blue represents a customer extension.
- A bold line with an arrow going from an entity back to itself means an edge foreign key.

See also

- “Accessing the Domain Graph” on page 282
- “Viewing the Domain Graph” on page 283

Object Ownership and the Domain Graph

The relationships that the domain graph captures is that of ownership. Thus, entity X owns entity Y if archiving entity X means that it is also necessary to archive entity Y. For example, in Guidewire ClaimCenter, Claim and

Matter hold this relationship. 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, or, in other words, which object owns the other object. In the ClaimCenter base configuration, there is a foreign key on Matter that points to Claim. This indicates that Matter is owned by Claim.

The following Matter metadata definition illustrates this.

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

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

For example, Claim has a foreign key to ClaimWorkComp. In this case, Claim owns ClaimWorkComp, not the reverse. In these cases, you must set the owner attribute on the foreign key to true. You must do this with extension entities as well, if you wish 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.

The following Claim definition description illustrates this.

```
<entity xmlns="http://guidewire.com/datamodel"
    desc="Insurance claim"
    entity="Claim" ... >
    ...
    <foreignkey columnName="ClaimWorkCompID"
        deletefk="true"
        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 290

Accessing the Domain Graph

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 151 in the *System Administration Guide*.

The Server Tools → Domain Graph Info page contains two tabs (technically known as cards). They are:

- **Graph**—Provides human-readable text versions of the domain graph. It also contains a **Download** button that generates a ZIP file that contains 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 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

Note: It is possible to add the path to the graphics software executable that you use to generate the .dot files to the PATH environment variable. If you do so, then the download .zip file includes a PDF version of the graphics file as well.

To view a domain graph visually, you need to 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 is Graphviz, which you can download from the following URL:

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

There is a **Download** link on the main page of this web site, at the left-hand side of the page. After you download and install the software, you can use it to generate a graphic representation of the domain graph.

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. For example, in Guidewire ClaimCenter, the file contains the following files:

```
ArchiveGraphInfo  
construction.log  
domain.dot  
domain.pdf  
index.html  
sortable.js
```

Note: 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 wish 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.
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 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	During the archiving process, ClaimCenter creates a stub or skeleton object that it leaves behind in the main database as it archives data. This object must implement the RootInfo delegate. IMPORTANT Only one object can implement the RootInfo delegate. For example, for ClaimCenter, the RootInfo object is ClaimInfo. You cannot change the RootInfo object.
All other domain objects	All domain objects—including the root object—must implement the Extractable delegate.
Reference objects	A reference object is a data object that multiple instances of a domain graph object all share. For example, multiple Claim 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. For example, the Address table is one such table. The database table itself exists in both the domain graph and as reference data. The primary use for these types of objects is for Guidewire code. <i>Their use by non-Guidewire code is not common.</i> All overlap table objects—and only overlap table objects—must implement the OverlapTable 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 284
- “Defining Foreign Keys Between Objects” on page 287

Implementing the Correct Delegate

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

- *The RootInfo Delegate*
- *The Extractable Delegate*
- *The OverlapTable Delegate*

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. Each delegate type provides additional columns on the affected tables.

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 a discussion of the Guidewire data model in general, see “The ClaimCenter Data Model” on page 187.
- For a discussion of delegate objects and how to work with them, see “Delegate Data Objects” on page 197.
- For a discussion of how to use the <implementsEntity> element, see “<implementsEntity>” on page 224.

The *RootInfo* Delegate

During the archiving of an instance of the domain graph, ClaimCenter leaves behind (in the main ClaimCenter database) a skeleton entity that provides the following:

- Sufficient information to restore 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 which entity implements the *RootInfo* delegate.

Thus, *ClaimInfo* implements the *RootInfo* delegate.

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

```

The *ClaimInfo* metadata defines the stub of the claim that ClaimCenter leaves behind after archiving a claim. This object contains minimal (summary) information about the claim. For example, it contains the claim number and loss location. ClaimCenter uses this information during claim search for archived claims.

IMPORTANT Because ClaimCenter does not archive the *ClaimInfo* table, it has the potential to grow very large. Guidewire recommends that you take care so that you do **not** put large amounts of data (for example, a BLOB) into the table if you extend it.

ClaimInfo Properties

The *ClaimInfo* object contains a number of properties. The following list describes some of the more useful properties. For more information on the *ClaimInfo* object, see “Archive” on page 48 in the *Rules Guide*.

Property	Description
<code>ClaimInfo.ArchiveState</code>	Each time a claim changes (including during archiving or restoration), ClaimCenter generates a <i>ClaimChanged</i> event. This event does not provide specific information about the archive state of the claim, however. Instead, to determine the archive state of a claim, use <i>ClaimInfo.ArchiveState</i> . This field can take the following values: <ul style="list-style-type: none">• <code>archived</code>• <code>retrieving</code> An <i>ArchiveState</i> of <code>null</code> indicates that the claim is in the active database, either because it has never been archived or because it has been successfully restored from the archive.
<code>ClaimInfo.ExcludedFromArchive</code>	(Boolean) If true, ClaimCenter excluded this claim from the set of claims to archive.

Property	Description
ClaimInfo.ExcludeReason	Use to see the reason that ClaimCenter excluded a claim from archiving.
ClaimInfo.PurgeDate	Use to set a date after which it is safe to purge the claim from the archive database. See "Archive Claim Purge Rules" on page 51 in the <i>Rules Guide</i> for a discussion of how to use the PurgeDate field with Gosu rules.

ClaimCenter generates an event if the `ArchiveState` of a `ClaimInfo` object changes to `archived`. Thus, you can listen for this event and generate a message (for an external document management system, for example) if ClaimCenter archives a claim.

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 illustrates the implementation of 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"/>
...

```

WARNING Any entity that is part of the domain graph **must** implement the `Extractable` delegate using the `<implementsEntity>` element. This means 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 add it manually, *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. The `Address` table is one such table. The `Address` table itself exists in both the domain graph and as reference data. However, individual rows in the table belong to (exist in) either the domain graph, or the reference data. Any attempt to create a table row that exists in both causes archiving to fail.

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.

As 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"/>
...

```

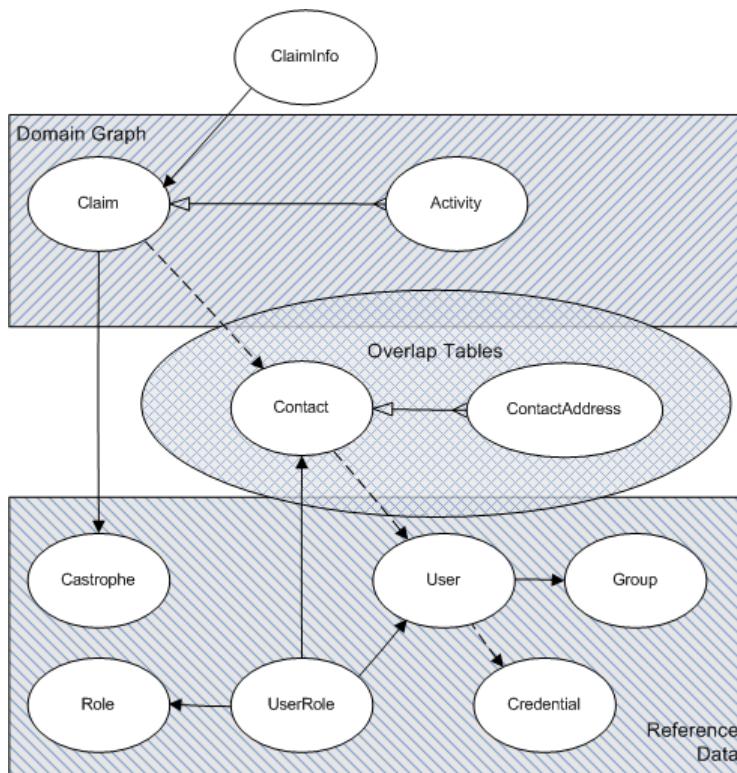
IMPORTANT It is not common for there to be a need to use overlap tables and thus the `OverlapTable` delegate. *It is primarily for use by Guidewire internal code.*

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 282, 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 <i>opposite</i> 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 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.

The following graphic, which shows portions of the objects in the ClaimCenter data model, illustrates the relationships between objects and foreign keys. The graphic 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, as 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. Recall that objects that implement the OverlapTable delegate can exist in the domain graph and as reference data. However, individual rows in the table can **only** exist in one table or the other

Graph Validation Checks

The ClaimCenter server performs a series of checks during startup. These checks verify that the current data model can support archiving, purging and export of claims. Some of these checks prevent the server from starting. Other checks are warnings and allow the server to start. These checks warn about potential problems with the graphs that might not be an issue depending on business logic.

The following table describes checks that prevent the server from starting. If any of these checks fails, the server reports an exception and prints out the cycle in DOT format. You can use a program such as GraphViz to view the graph.

Check	Description
Domain graph not partitioned	Verifies that all domain graph tables are reachable through the root entities.
Edge tables in domain graph have required foreign keys	Verifies that if edgeTable is set on an entity in the domain graph, it must have all of the following: <ul style="list-style-type: none">An owned foreign key back to one of its parentsAn unowned foreign key to that same parent.
No cycles in domain graph	<p>Looks for circular references in the domain graph. The domain graph is the set of tables and their relationships that define an archival unit, such as a Claim, its Exposures and Contacts, and so forth.</p> <p>The archiving and purging processes move non-shared data related to a Claim to the archive by traversing the domain graph. Circular references in the graph cause issues for this process.</p> <p>The domain graph cannot have any cycle in its "is owned by" relationships. Thus, the following example fails validation: A "is owned by" B "is owned by" C "is owned by" A. You need to resolve any cycles by sorting out the ownership relationships.</p>
Domain graph entities implement Extractable	Ensures that all entities inside the domain graph implement the Extractable delegate as ClaimCenter uses the following columns on Extractable during the archive process: <ul style="list-style-type: none">ArchivePartitionArchiveID This check also verifies that no entities outside of the domain graph implement the Extractable delegate.
Overlap tables implement OverlapTable	Verifies that all overlap tables implement the OverlapTable delegate. An overlap table contains rows that can exist either in the domain graph or as part of reference data, but not both. Overlap tables must implement the following delegates: <ul style="list-style-type: none">ExtractableOverlapTable
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 retrievable	Verifies that all reference entities in the domain graph points are retrievable.
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 check verifies that the outside entity is indeed outside the domain graph.

If any of these checks finds an issue, ClaimCenter prevents the server from starting and prints the graph in DOT notation. You can use the DOT format graph output to view the graph using graph visualization software such as GraphViz.

In addition, the **Archive Info** page provides warnings for situations that could potentially lead to errors. See “Domain Graph Info” on page 161 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 checks.

Check	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. However, the existence of such foreign keys does not cause outright failure as it is possible for an archiving rule to prevent the archiving of such graph instances.
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 archiving or purging process is not able to tag the correct entities. This check 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 only 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...

In this case:

1. If a graph never referred to that entity, it does not appear in the XML. There is no issue.
2. If a graph that has been archived referred to an instance of that entity, it appears as a referenced entity in the XML. This means on restore, 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 it was some piece of reference data that was shared by multiple claims. It 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, however, 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. (The entity was needed for some reason, for example.) On restore, the archived instance then causes a duplicate key violation as 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. For example, you cannot have two claims that reference the same Event entity. Guidewire does not allow this in archiving as it violates the delimitation of a unit of work. However, there are cases in which you want to share data across multiple claims. 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 (for example, an Event) across multiple claims, you can extend the `ClaimInfo` object and add a foreign key to the entity table. However, if you choose to do this, 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. This means, for example, that not all of the related claims that the code references can exist in the main database.

If you intend to extend the `ClaimInfo` object and 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. This means that 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 problem occurs if A has a foreign key to B and B has a foreign key to A. In this situation, 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.

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

way to indicate this relationship is to add the special `owner="true"` attribute to the foreign key to make the ownership clear.

IMPORTANT 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 282

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 293
- “Field Validator Definitions” on page 294
- “Modifying Field Validators” on page 297
- “Field Validation and Localization” on page 298

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.

Note: For complex validation between fields, use validation-specific Gosu code instead of simple field validators. For more information on validation, see the relevant validation section in the *ClaimCenter Rules Guide*.

Specifying 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}([x0-9]{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.

IMPORTANT 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 on the <ValidatorDef> element specifies the name of the validator. A field definition uses this attribute to specify which validator applies to the field.

Value

The value attribute on the <ValidatorDef> element specifies the acceptable values for a String field. It is in the form of a regular expression. 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>

ClaimCenter does not persist the regular expression definition to the database.

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 on the `<ValidatorDef>` element 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. For example, the display description for the `Validator.Money` key is:

```
Validator.Money = {0} must be a money value
```

Input-Mask

The `input-mask` attribute on the `<ValidatorDef>` element 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 on the `<ValidatorDef>` element works in a similar manner to the `input-mask` attribute. However, it is not currently in use.

Placeholder-Char

The `placeholder-char` attribute on the `<ValidatorDef>` element 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 on the `<ValidatorDef>` element 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="99999999999999.99"
               floor="-99999999999999.99"
               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>
```

Note: It is important to understand that 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.

Field Validation and Localization

It is possible to create multiple country-specific field validator files. See “Localizing Field Validators” on page 507 for details.

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 299
- “The ‘datatypes.xml’ File” on page 302
- “Customizing Base Configuration Data Types” on page 303
- “Working with Money and Currency Data Types” on page 304
- “Working with the Medium Text Data Type (Oracle)” on page 308
- “The Data Type API” on page 308
- “Defining a New Data Type: Required Steps” on page 310
- “Defining a New Tax Identification Number Data Type” on page 310

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 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 type (defined in the `valueType` attribute). For example, the associated type for the `datetime` data type is `java.util.Date`. Thus, you see in the `datetime` definition file, the following:

```
<DataTypeDef xmlns="http://guidewire.com/datatype"
    type="com.guidewire.pl.metadata.datatype2.impl.DateTimeDataTypeDef"
    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.DecimalDataTypeDef"
    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 <code>datatypes.xml</code> , which you access through Studio. You can only modify a select subset of the base configuration data types. See “Customizing Base Configuration Data Types” on page 303.
Create a new data type	Create a <code>.dti</code> definition file and place it in <code>ClaimCenter/modules/configuration/config/datatypes</code> . You also need to create Gosu code to manage the data type. See “Defining a New Data Type: Required Steps” on page 310.
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 <code>VARCHAR</code> column in the base data model use encryption by extending the entity and setting the <code>encryption</code> parameter on a <code><columnParam></code> element. See “ <code><column></code> ” on page 212 for a discussion of the <code><columnParam></code> subelement.

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, 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>
```

WARNING 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. Guidewire recommends that your company database administrator (DBA) determine the maximum row size and how to 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. 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`

Note: There can be multiple config/datatypes folders in your application instance. To determine the complete set of Guidewire data types, you must view the files in all locations.

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 that 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 *only* associate a data type with a property whose feature type matches the data type of the value type. For example, you can only associate a `String` data type with `String` properties.

IMPORTANT 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 'datatypes.xml' 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 permits you to modify certain attributes on a small subset of the base configuration data types. It lists these data types in file `datatypes.xml`, which you can access in Studio at **Resources → Data Model Extensions**. You can modify the values of certain attributes in this file to customize how these data types work in ClaimCenter.

This 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 Do **not** modify this file unless you know exactly what you are doing. You must also use Studio to manage this file. If you modify this 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, `<MoneyDataType>`. This 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 DECIMAL types only. <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. Therefore, <code>precision >= scale</code> (always).
<code>app</code>	Optional attribute for use with money data types. See "Money Data Types" on page 305 for information.

Deploying Changes to `datatypes.xml`

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. 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.

IMPORTANT *ClaimCenter only looks at the data type definitions 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 303. 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. If you are working with monetary values, see “Working with Money and Currency Data Types” on page 304. There are special requirements for these attributes in working with monetary amounts.

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, see “Field Validation” on page 293.
- For information on how to localize field validation, see “Localizing Field Validators” on page 507.

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: 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 >=scale (always).

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
VIN	VARCHAR	length, validator
Year	INTEGER	validator

The PercentageDec Data Type

Guidewire builds this data type on top of the DECIMAL (3,0) data type. Only use decimal values from 0 to 100 inclusive.

Working with Money and Currency Data Types

In the base configuration, Guidewire defines multiple data types that apply to money or currency. You use these data types to store and display monetary values within Guidewire ClaimCenter. Some of these data types you use to implement single currency configurations. Others, Guidewire specifically designs for use in implementing multicurrency configurations. In general, you can divide the monetary data types into two categories.

Setting the Currency Mode

IMPORTANT Unless you are installing and configuring Guidewire ClaimCenter, set the currency mode to **SINGLE**. If you are installing Guidewire ClaimCenter, consider carefully the differences between the two modes.

The currency mode determines how ClaimCenter stores money values and displays them in the interface. You use configuration parameter `MultiCurrencyDisplayMode` in `config.xml` to set the currency mode for the application. The currency mode is either *single currency* or *multicurrency*. The valid configuration parameter values are:

- **SINGLE**
- **MULTIPLE**

The decision of which currency mode to implement is based on whether you want to see multicurrency features in the ClaimCenter interface. For example, **SINGLE** mode hides multicurrency fields such as `Currency` and `ExchangeRate` fields, as they are irrelevant.

Currency mode	Recommendations
SINGLE	<p>In SINGLE currency mode, use <code>money</code> data type for money amount columns if you do not foresee moving to a multicurrency implementation. In this case, you are using only one currency and there is no possibility of any confusion.</p> <p>However, there are some advantages of using the <code>currencyamount</code> data type, even in SINGLE mode:</p> <ul style="list-style-type: none">• Using the <code>currencyamount</code> data types provides currency-safety as it is not possible to mistakenly add currency amounts with different currencies.• It is possible to write Gosu code such that it passes <code>CurrencyAmount</code> values for monetary amounts. You need to make this change in any case if you switch to MULTIPLE currency mode at a later time.
MULTIPLE	<p>In MULTIPLE currency mode, always use the <code>currencyamount</code> data type and always pass around <code>CurrencyAmount</code> values for monetary amounts. You can change the type on <code>money</code> extension columns to be <code>currencyamount</code> columns without any upgrade cost. (This change does trigger a database upgrade. However, the upgrade utility does not actually make any changes to the database due to the type change.)</p>

See Also

- “Configuring Currency” on page 623
- “Multiple Currencies” on page 171 in the *System Administration Guide*

Money Data Types

Use the `money` data type to store and show money values in the system. In a single currency configuration, Guidewire ClaimCenter assumes all money amounts in a `money` data type column are in the same currency. The following list describes the Guidewire base configuration `money` data types:

Data type	Description
<code>money</code>	Permits positive, negative, and zero values.
<code>nonnegativemoney</code>	Does not permit negative values. However, zero values are acceptable.
<code>positivemoney</code>	Does not permit negative or zero values.

The `money` data types use the following specialized attributes, which you set in `datatypes.xml`:

- `precision`, `scale`
- `appscale`

These settings for these parameters affect all `money` columns.

precision, scale

With money values, `precision` controls the total number of digits and `scale` controls the number of digits to the right of the decimal point. For example, if `precision=5` and `scale=2`, then the maximum value for money is 999.99 and the minimum value is -999.99.

Some factors to consider in choosing the `scale` value include localization issues and the needs of the specific industry. For example, if it is important to track money amounts down to 1/100 of a cent, then set `scale` to 4. (The default is 2.)

Guidewire recommends that you set the `scale` attribute to the maximum that you would possibly need in the future. This is useful if you decide to support additional currencies in the future that could use a number of decimal digits larger than the current configuration.

You set `precision` and `scale` in the `<MoneyDataType>` element in file `datatypes.xml`. (See also “The ‘datatypes.xml’ File” on page 302.)

For example:

```
<MoneyDataType precision="18" scale="2" validator="Money"/>
```

IMPORTANT The `precision` and `scale` attributes control both how ClaimCenter displays money values in the interface and how ClaimCenter stores money values in the database.

WARNING You cannot change the `scale` attribute without dropping the database.

appscale

The `appscale` attribute is a rarely-used optional data type attribute. It enables a single-currency installation to specify a scale appropriate to that particular currency, while the database column that stores the amount uses a larger scale attribute value. The `appscale` becomes the effective scale of `BigDecimal` values in the application that have been loaded from `money` and `currencyamount` columns in the database. The `appscale` value must be less than the `scale` value in use in the database.

Guidewire provides this attribute to facilitate an eventual upgrade to a multicurrency configuration that will make use of the full `scale` values. As an example, suppose that you implement a Japanese Yen (0-`scale`) installation that you intend to become a multicurrency installation in the future.

The future multicurrency installation needs to support currencies with a larger scale, such as U.S. Dollar (2-`scale`). If you know this at the time of the original installation, then you want the database column to handle 2-`scale` values in the initial set up. Otherwise, you must recreate those columns later to have 2-`scale`, which the Guidewire upgrade tool does not currently support.

In this example, you would initially set `appscale=0` and `scale=2`. Thus, all `BigDecimal` values in the application for `money` and `currencyamount` columns would have a scale of 0. Later, after you transition the installation to a multicurrency installation, you would remove the `appscale` attribute. In its place, you would use the `storageScale` attributes in `currencies.xml` instead to specify the `scale` to use for each currency.

The following rules and restrictions apply to the `money` data type.

- If you do not specify an `appscale` attribute, then ClaimCenter uses the `scale` attribute value.
- If you do specify an `appscale` value, make the `appscale` value smaller than the `scale` value.

- If you do specify an `appscale` value, set the value to the largest decimal value required for the currently configured currency.

IMPORTANT The `appscale` attribute does not exist in the base configuration. You need to add this attribute to the `<MoneyDataType>` element in file `datatypes.xml` if you want to use it. This attribute is only useful in the SINGLE currency mode. In MULTIPLE currency mode, use the `storageScale` attribute on each currency listed in `currencies.xml`.

Currency Amount Data Types

Guidewire provides a number of currency data types that extend the `money` data types to specify the currency of the money amount being stored. The primary purpose of the `currencyamount` data types is for use in multicurrency configurations.

The following list describes the Guidewire base configuration `currencyamount` data types.

Data type	Description
<code>currencyamount</code>	Permits positive, negative, and zero values.
<code>nonnegativecurrencyamount</code>	Does not permit negative values. However, zero values are acceptable.
<code>positivecurrencyamount</code>	Does not permit negative or zero values.

In both Gosu and Java code, a `currencyamount` property returns a `CurrencyAmount` object, which associates a `BigDecimal` money amount with a particular `Currency` typekey value. ClaimCenter persists the `BigDecimal` amount to the database. You can configure the `Currency` value for that particular column.

To specify the currency of the amount that you are storing in a column, use the `<columnParam>` element on that column to set a `currencyProperty` value. The `currencyProperty` parameter points to a property on the entity that returns the `Currency` for the column. This property can be either of the following:

- A virtual property, which you typically define in an enhancement class
- A `Currency` column on the entity, which stores a `Currency` typekey value

If the property is a virtual property, it usually looks up the relevant `Currency` from a parent entity, such as a `Claim`.

For example, in ClaimCenter, if you want to store a money amount in the same currency as the claim, then you can use `ClaimCurrency` as the currency property.

```
<column name="SomeAmount" type="currencyamount" ...>
  <columnParam name="currencyProperty" value="ClaimCurrency"/>
</column>
```

Note: See also “Multiple Currencies” on page 171 in the *Application Guide*.

In the base configuration, Guidewire provides a definition for the `ClaimCurrency` property on many entities, either in Java or through a Gosu enhancement class. If it does not exist, you can define it in an enhancement class on that particular entity type.

It is possible to create a column that contains a multicurrency value that could potentially contain any `Currency`. In this case, you need to define another extension column on the entity to store the `Currency` value and reference that in the `currencyProperty` `<columnParam>` element. For example:

```
<typekey name="SomeCurrency" nullok="false" typelist="Currency" ...>
<column name="SomeAmount" type="currencyamount" ...>
  <columnParam name="currencyProperty" value="SomeCurrency"/>
</column>
```

It is not mandatory to provide the `currencyProperty` in a `<columnParam>`. If you do not, then ClaimCenter defaults to using the value that you set for `DefaultApplicationCurrency` in `config.xml`.

See Also

- “<column>” on page 212
- “The <columnParam> Subelement” on page 214

Setting the Currency Display

Guidewire ClaimCenter uses the following to determine the number of decimal places to show in the interface:

Currency mode	Decimal digits to show...
SINGLE	<p>ClaimCenter uses the value of the <code>appscale</code> attribute on the <code>money</code> data type in <code>datatypes.xml</code> to determine the number of decimal digits to show in the interface. If you do not set a value for <code>app</code>, then ClaimCenter uses the <code>scale</code> value to determine the number of decimal digits to show.</p> <p>IMPORTANT The <code>appscale</code> attribute does not exist in the base configuration. You need to add this attribute to the <code><MoneyDataType></code> element in the <code>datatypes.xml</code> file.</p>
MULTIPLE	<p>ClaimCenter uses the value of the <code>storage</code> attribute on <code><CurrencyType></code> in <code>currencies.xml</code> to determine the number of decimal digits to show in the interface.</p> <p>The <code>appscale</code> attribute is useful in multicurrency configurations for the <code>money</code> data types. However, with <code>currencyamount</code> data types, the <code>storage</code> parameter supersedes the <code>appscale</code> attribute on the <code><MoneyDataType></code> element in <code>datatypes.xml</code>. Make the currency <code>storage</code> value less than or equal to the <code>appscale</code> value and make the <code>appscale</code> value less than or equal to the <code>scale</code> value. The following must always be true.</p> $\text{storage} \leq \text{appscale} \leq \text{scale}$

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. 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.

The following sections discuss:

- 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 that takes all parameters
- One 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.

IMPORTANT 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:

- a. Select the **Other Resources** folder, right-click, and select **New → Other file**.
- b. 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.
- c. Click the file name in the **Resource tree**. Studio opens the file for editing.

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 299.

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 prop-

erty, the name of the property on the context object. This property—called `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. Select **Other Resources**, right-click and select **New → Other file**.
2. Enter `TaxID.dti` as the file name. This action creates an empty data type file and places it in the `datatypes` folder.
3. Enter the following text in the file:

```
<?xml version="1.0"?>
<DataTypeDef xmlns="http://guidewire.com/datatype" type="gw.newdatatype.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	TaxID
value type	String
parameter	<code>contactType</code>
implementation type	<code>gw.newdatatype.TaxIDDataTypeDef</code>

See Also

- For details on the attributes and elements relevant to the data type definition, see “The ClaimCenter Data Model” on page 187.

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.I PropertyInfo
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 : I PropertyInfo, errors : IDataTypeDefValidationErrors) {
        // Check that the CountryProperty names an actual property on the owning type, and that
        // the type of the property is typekey.Country.
        var countryProp = prop.OwnersType.TypeInfo.getProperty(CountryProperty)

        if (countryProp == null) {
            errors.addError("Property '" + CountryProperty + "' does not exist on type " +
                           prop.OwnersType)
        } else if (not typekey.Country.Type.isAssignableFrom(countryProp.Type)) {
            errors.addError("Property " + countryProp + " does not resolve to a " + typekey.Country)
        }
    }
}
```

Note that the class defines a property named `CountryProperty`, which the system calls to pass the `countryProperty` parameter. Also notice how the implementation reads the value of `CountryProperty` as its constructs its constraints and presentation handlers. Guidewire guarantees to fill the implementation parameters before calling the handlers.

In the example code, the class refers to constraints and presentation handlers created specifically for this data type. However, it also reuses a Guidewire-provided persistence handler, the `VarcharPersistenceHandler`. You do not usually need to create your own persistence handler, as Guidewire defines persistence handlers for all the basic database column types.

Step 3: Implement the Data Type Aspect Handlers

As you define a new data type, it is possible (actually likely) that you need to define one or more handlers for the data type. These handler interfaces are different than the Data Type API interfaces. For example, clients that use the Data Type API use the following:

```
gw.datatype.IConstrainedDataType
```

However, if you define a new data type, you must implement the following:

```
gw.datatype.handler.IDataTypeConstraintsHandler
```

This separation of interfaces allows the definition of a caller-friendly interface for data type clients and a implementation-friendly interface for data type designers.

The example data type defines a handler for both constraints and presentation.

Class TaxIDConstraintsHandler. This class looks similar to the following:

```
package gw.newdatatypes

uses gw.datatype.handler.IStringConstraintsHandler
uses gw.lang.reflect.IPropertyInfo
uses java.lang.Iterable
uses java.lang.Integer
uses java.lang.CharSequence
uses gw.datatype.DataTypeException

class TaxIDConstraintsHandler implements IStringConstraintsHandler {

    var _countryProperty : String

    construct(countryProperty : String) {
        _countryProperty = countryProperty
    }

    override function validateValue(ctx : Object, prop : IPropertyInfo, value : Object) {
        var country = getCountry(ctx)

        switch (country) {
            case "US": validateUSTaxID(ctx, prop, value as java.lang.String)
                break
            // other countries ...
        }
    }

    override function validateUserInput(ctx : Object, prop : IPropertyInfo, strValue : String) {
        validateValue(ctx, prop, strValue)
    }

    override function getConsistencyCheckerPredicates(columnName : String) : Iterable<CharSequence> {
        return {}
    }

    override function getLoaderValidationPredicates(columnName : String) : Iterable<CharSequence> {
        return {}
    }

    override function getLength(ctx : Object, prop : IPropertyInfo) : Integer {
        var country = getCountry(ctx)

        switch (country) {
            case "US": return ctx typeis Person ? 11 : 10
            // other countries ...
        }

        return null
    }

    private function getCountry(ctx : Object) : Country {
        return ctx[_countryProperty] as Country
    }

    private function validateUSTaxID(ctx : Object, prop : IPropertyInfo, value : String) {
        var pattern = ctx typeis Person ? "\d{3}-\d{2}-\d{4}" : "\d{2}-\d{7}"
        if (not value.matches(pattern)) {
            throw new DataTypeException("${value} does not match required pattern ${pattern}", prop,
                "Validation.TaxID", { value })
        }
    }
}
```

Class TaxIDPresentationHandler. This class looks similar to the following:

```
package gw.newdatatypes

uses gw.lang.reflect.IPropertyInfo
```

```
uses gw.datatype.handler.IStringPresentationHandler

class TaxIDPresentationHandler implements IStringPresentationHandler {

    private var _countryProperty : String

    construct(countryProperty : String) {
        _countryProperty = countryProperty
    }

    function getEditorValue(ctx : Object, prop : IPropertyInfo) : Object {
        return null
    }

    override function getDisplayFormat(ctx : Object, prop : IPropertyInfo ) : String {
        return null
    }

    override function getInputMask(ctx : Object, prop : IPropertyInfo) : String {

        switch (getCountry(ctx)) {
            case "US": return ctx typeis Person ? "###-##-####" : "##-#####"
            // other countries ...
        }

        return null
    }

    override function getPlaceholderChar(ctx : Object, prop : IPropertyInfo) : String {
        return null
    }

    private function getCountry(ctx : Object) : Country {
        return ctx[_countryProperty] as Country
    }
}
```

Notice how each of these handlers makes use of the context object in order to determine the type of input mask and validation string to use.

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 315
- “Terms Related to Typelists” on page 316
- “Typelists and Typecodes” on page 316
- “Typelist Definition Files” on page 317
- “Different Kinds of Typelists” on page 317
- “Working with Typelists in Studio” on page 318
- “Typekey Fields” on page 322
- “Typelist Filters” on page 325
- “Static Filters” on page 325
- “Dynamic Filters” on page 330
- “Dynamic Filters” on page 330
- “Mapping Typecodes to External System Codes” on page 333

What is a Typelist?

WARNING Guidewire recommends that you fully understand the dependencies between typelists before you modify one. 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 type-list.

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 type-codes (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

See “Localizing Typecodes” on page 486 for information on how to localize each individual typecode in a type-list.

Mapping Typecodes to External System Codes

See the following:

- “[Mapping Typecodes to External System Codes](#)” on page 333
- “[Mapping Typecodes to External System Codes](#)” on page 81 in the *Integration Guide*

Typelist Definition Files

Similar to entity definitions, Guidewire ClaimCenter 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 Object Files](#)” on page 189

Different Kinds of Typelists

ClaimCenter organizes typelists into the following categories:

Category	Description
Internal	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. You can, however, override the following attribute values on these types of typelists: <ul style="list-style-type: none">• name• description• priority• retired
Extendable	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 these typecodes, but you can modify any of the example typecodes.
Custom Typelists	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).

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.

The following are examples of internal typelists that you cannot change:

- `ActivityType`
- `CancellationTarget`
- `ChargePattern`
- `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 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 238.

To create a custom typelist, select **Typelists** from the **Resources** tree, then right-click and select **New → Typelist**. Enter a name for the typelist, then define your typecodes.

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 **Typelists** folder in the Studio **Resources** tree 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 **Typelists** editor.
- To create a new *custom* typelist, select **Typelists**, then right-click and select **New → Typelist**. After entering its name, you can 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.

IMPORTANT Studio manages the XML files that define the ClaimCenter typelists. *Guidewire expressly does not support working directly within an XML typelist file*. To create, modify, or extend a typelist, use Guidewire Studio only.

The Typelists Editor

WARNING If you modify an existing typelist, Guidewire recommends 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 **Resources → Typelists** folder, Studio opens a typelist editor showing configuration options for that typelist.

The Studio Typelists Editor Interface

The top portion of the **Typelists** 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.

IMPORTANT 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 typelist 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	cctl_typeList-name
Table name	Typelists editor	25 characters	cctl_table-name

IMPORTANT Studio does not enforce the 25 character limit in the **Table name** field. However, if you enter a character string in this field that is longer than 25 characters, the application server refuses to start.

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 **Typelists** 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 321
Filters	Define a fixed subset of a typelist to use as a static filter.	• “Static Filters” on page 325
Categories	Create a typelist filter that depends on the typecodes in a different typelist.	• “Dynamic Filters” on page 330
Localizations	Assign a locale to a typecode and enter translation strings for name and description.	• “Localizing Typecodes” on page 486

Note: For information on working with the **Localization** tab of the **Typelists** editor, see “Localizing Typecodes” on page 486.

To create a new typelist

- Select **Typelists** in the Resources tree.
- Right-click and select **New → Typelist**.
- Enter the typelist name in the **New Typelist** dialog. ClaimCenter uses this name to uniquely identify this typelist in the data model.

Note: Guidewire limits the length of the typelist name to a maximum of 25 characters, unless you enter a value in the **Table name** field.

- 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*.
- 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.

For this field:

- True* means that you cannot add or delete typecodes from the typelist. You can only override certain attribute fields.

- *False* means that 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.)

IMPORTANT Studio does not propagate typelist modifications (additions or customizations) to the application server. ClaimCenter regards any change that you make to a typelist as a change to the data model. You must stop and restart the application server to have ClaimCenter pick up your changes.

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 (:) <p><i>Do not include white space or use a hyphen (-).</i></p> <p>Use this code to map to your legacy systems for import and export of ClaimCenter data. The code must be unique within the list. Guidewire limits typecodes to 50 characters in length.</p> <p>See also “Mapping Typecodes to External System Codes” on page 333.</p>
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> .
	IMPORTANT You must enter a typecode description if you add a new typecode. If you do not: <ul style="list-style-type: none">• Studio generates error messages but continues to start.• The application server generates error messages and refuses to start.
Priority	A value that determines the sort order of the typecodes (lowest 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).

IMPORTANT ClaimCenter Standard Reporting can contain drill-down reports. These are reports that provide hyperlinks on the claim number that you click to access claim information directly within ClaimCenter itself. However, passing a typelist as a parameter to a drill-down report can cause issues if one of its typecode names contains a comma. For details, see “Configuring Drill-Down Reports” on page 84 in the *Reporting Guide*.

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 need 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.

WARNING There is an upper limit of 8000 typecodes on a typelist (due to a Java limitation). If you attempt to create a typelist with a larger number of typecodes, then Guidewire ClaimCenter refuses to compile.

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 a 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 316
- “Mapping Typecodes to External System Codes” on page 333
- “Typecodes and Web Services” on page 56 in the *Integration Guide*
- “Mapping Typecodes to External System Codes” on page 81 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 **Typelists** 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 318.

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" />
```

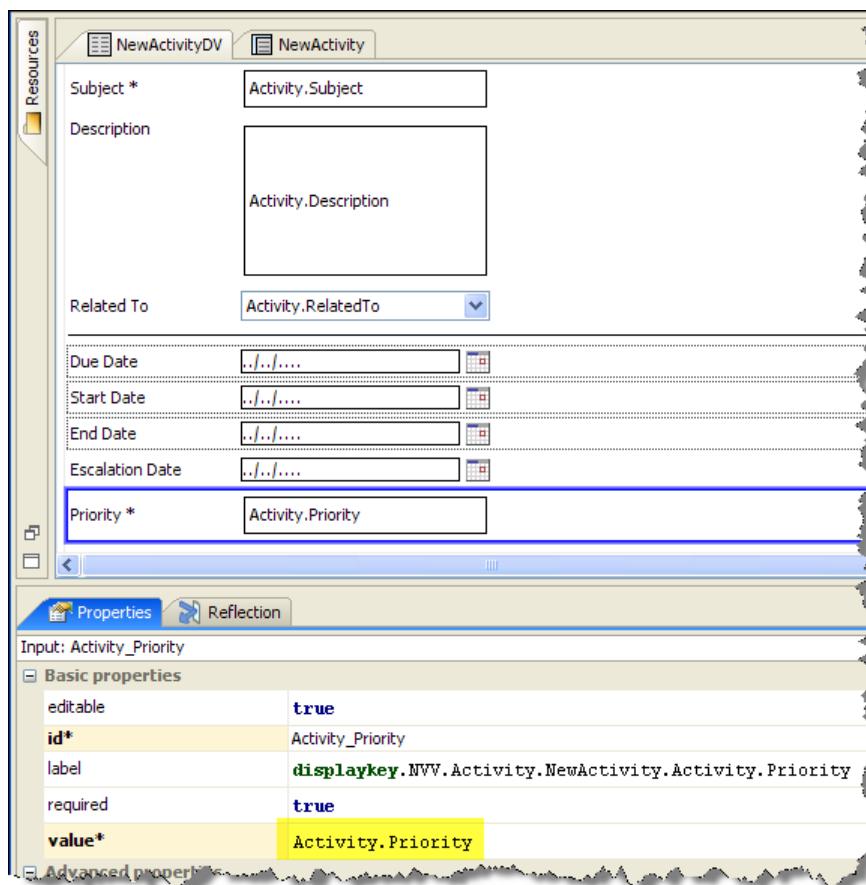
See Also

- For information on the `<typekey>` element, see “`<typekey>`” on page 229.
- For information on how to create data model entities, see “The ClaimCenter Data Model” on page 187.
- For information on how to modify existing data model entities, see “Modifying the Base Data Model” on page 233.

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 337.
- For information on working with PCF files in general, see “Introduction to Page Configuration” on page 349.

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.

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"> • <i>Include</i> certain specific typecodes on the typelist only. • <i>Include</i> certain specific <i>categories</i> of typecodes on the typelist. • <i>Exclude</i> certain specific typecodes from the full list of the typecodes on the typelist 	"Static Filters" on page 325
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 330

Static Filters

A *static* typelist filter causes the typelist to display only a subset of the typecodes for that typelist. Therefore, a static filter narrows the list of typecodes to show in the typelist view in the application. Guidewire calls this kind of typelist filter a static *typefilter*.

You define a static filter at the level of the typelist. You do this through the Studio **Typelists** editor, by defining a filter on the **Filters** tab for that particular typelist.

Studio manages the typelist XML file for you automatically. If you examine this file, you see that Studio uses the following XML syntax to define a static typelist filter. (In this case, a static filter that defines—or includes—a subset of the available typecodes.)

```
<typelistextension xmlns="http://guidewire.com/typelists" desc="Yes, no or unknown" name="YesNo">
  <typecode code="No" desc="No" name="No" priority="2"/>
  <typecode code="Yes" desc="Yes" name="Yes" priority="1"/>
  <typecode code="Unknown" desc="Unknown" name="Unknown" priority="3"/>
  <typefilter desc="Only display Yes and No typelist values" name="YesNoOnly">
    <include code="Yes"/>
    <include code="No"/>
  </typefilter>
</typelistextension>
```

Notice that the XML declares each typecode on the typelist (Yes, No, and Unknown). It then specifies a filter named YesNoOnly that limits the available values to simply Yes and No. This is static (fixed) filter.

Note: For more information on the `<typefilter>` element, see “`<typekey>`” on page 229.

To create a static filter

1. Define the typecodes for this typelist in the Studio **Typelists** editor. See “Working with Typelists in Studio” on page 318 for details.
2. Select the **Filters** tab on this typelist in the **Typelists** 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.
Description	Description of the context for which to use this typefilter.

Attribute	Description
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 typelist view in the application.

4. Use the fields in the following panes on the **Filters** tab to create a fixed subset of the typecodes for use in the static filter.

Subpane	Use to...	See...
Categories	Specify one or more typecodes to <i>include by category</i> within the filtered typelist view.	"Creating a Static Filter Using Categories" on page 326
Includes	Specify one or more typecodes to <i>include</i> within the filtered typelist view.	"Creating a Static Filter Using Includes" on page 328
Excludes	Specify one or more typecodes to <i>exclude</i> from the full list of typecodes for this typelist.	"Creating a Static Filter Using Excludes" on page 329

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 322 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

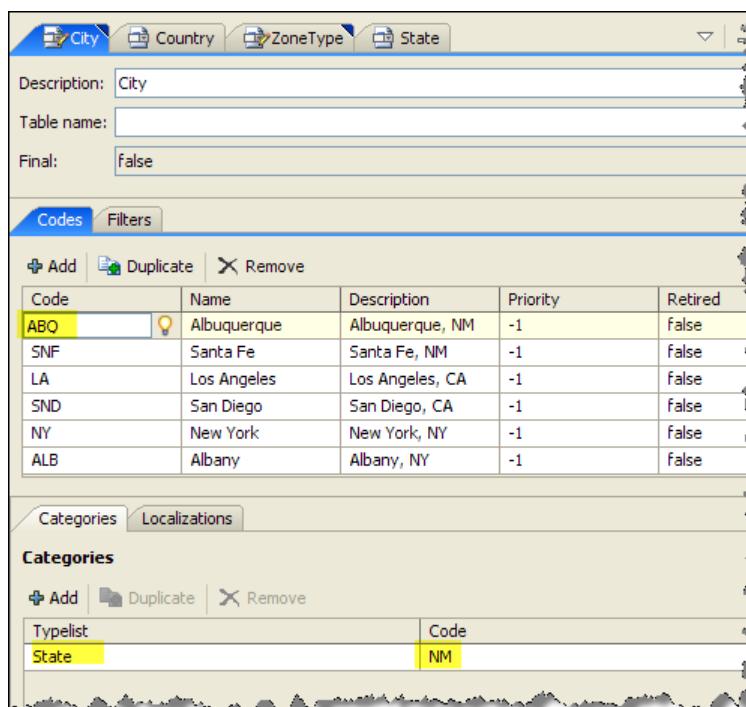
Suppose that you want to filter a list of cities by state. (Say that you want to only show a list of appropriate cities if you select a certain state.) To create this filter, you need to first to define a *City* typelist (if one does not exist). The following table lists a few sample cities:

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

After making your choices, you have something that looks similar to the following:



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 322 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 (for some reason) that you want to create a filtered typelist that displays zone codes that are only in use in Canada and not any other country. One way to create the filter, if there is a limit to the number of choices, is to use an **Includes** filter on the **ZoneTypes** typelist.

In this example, you only want the typelist to display 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. The top section is the 'Filters' pane, which contains a table with three columns: Name, Description, and Include All?. There are three rows: 'AustraliaFilter' (Description: 'Displays zone codes for', Include All?: false), 'CAOnlyFilter' (Description: 'Displays unique zone cc', Include All?: false, highlighted with a yellow background), and 'ExcludesCanada' (Description: 'Displays zone codes not', Include All?: true). Below the filters is the 'Includes' pane, which has a table with a single column 'Code'. It contains two rows: 'fsa' and 'province', with 'province' also having a yellow background.

Creating a Static Filter Using Excludes

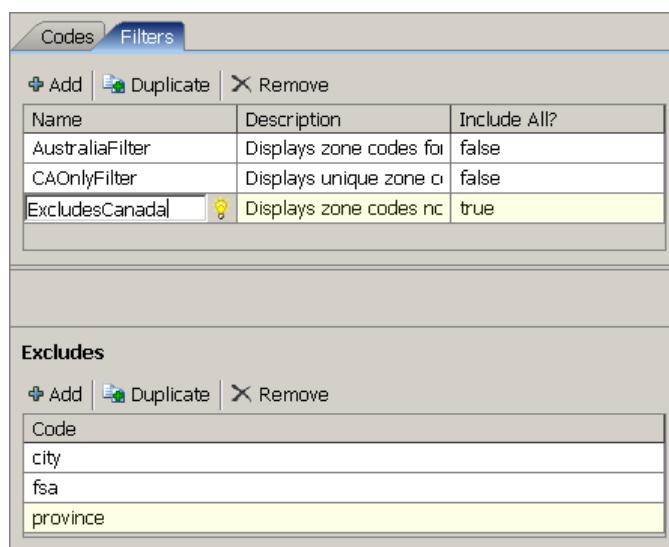
Suppose (for some reason) that you want to create a filtered typelist that displays all of the zone codes except those that are in use in Canada. You want to display the complete list of typecodes *except* for the following:

- 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.



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 by using the Studio **Typelists** editor to define a filter on the **Codes** tab for a particular typecode. To create this filter, you select a specific typecode and set a filter, a *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 **Typelists** 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:

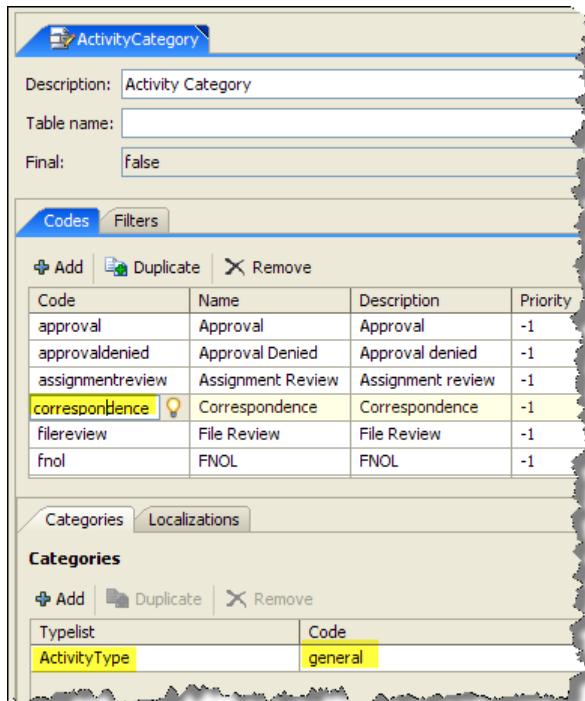
- Correspondence
- File Review
- General
- Interview
- ...

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

Note: 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>
```

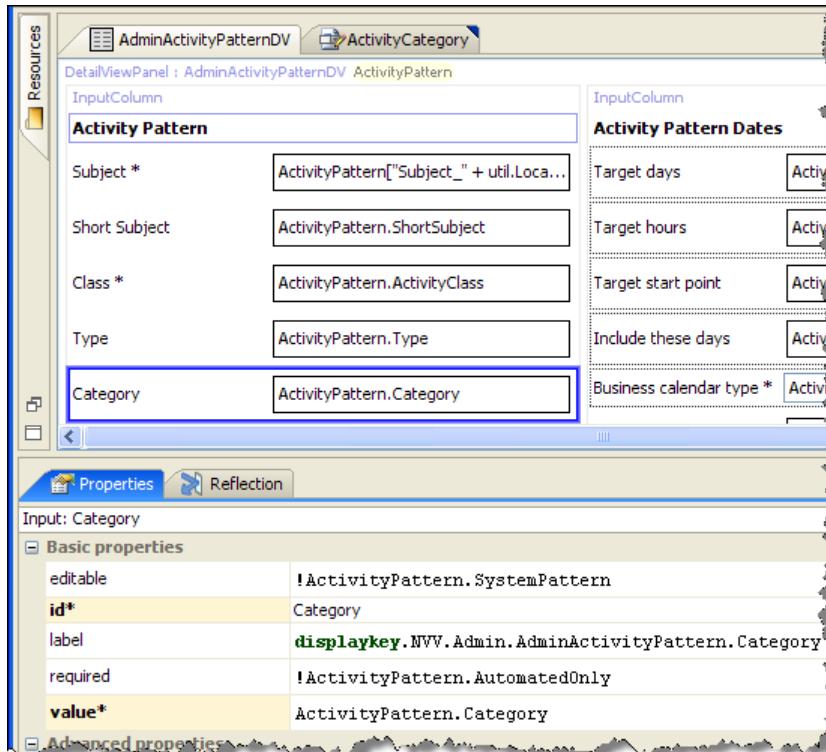
The sample code first defines a `<typekey>` element with `name="Type"` and `typelist="ActivityType"`. This is the controlling (parent) typelist. The code then defines a second typelist (`ActivityCategory`) with a keyfilter `name="Type"`. It is the typelist referenced by the `<keyfilter>` element that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

Note: For more information on the `<keyfilter>` element, see “`<typekey>`” on page 229.

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.

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 **Other Resources** folder.) This file specifies a *namespace* for each external application. Then, you identify the individual unique typecode maps by typelist.

A typecodemapping.xml example. The following code sample illustrates a simple `typecodemapping.xml` file:

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

  <typelist name="AccountSegment">
    <mapping typecode="PR" namespace="accounting" alias="ACT" />
  </typelist>
</typecodemapping>
```

The `namespacelist` tag contains one or more `namespace` tags—one for each external application. Then, to map the actual codes, you specify one or more `typelist` tags as required. Each `typelist` tag refers to a single internal or external typelist in the application. The `typelist`, in turn, contains one or more `mapping` tags. Each `mapping` tag must contain the following attributes:

- `typecode` Specifies the ClaimCenter typecode.
- `namespace` Specifies the name space to which ClaimCenter maps the typecode
- `alias` 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 81 in the *Integration Guide*.

User Interface Configuration

Using the PCF Editor

This topic covers how you work with PCF (Page Configuration Format) files in Guidewire Studio.

This topic includes:

- “The Page Configuration (PCF) Editor” on page 337
- “The PCF Canvas” on page 338
- “Creating a New PCF File” on page 339
- “The Toolbox Tab” on page 340
- “The Structure Tab” on page 340
- “The Translations Tab” on page 340
- “The Properties Tab” on page 341
- “PCF Elements” on page 343
- “Working with Elements” on page 343

The Page Configuration (PCF) Editor

Note: You can access the *Guidewire ClaimCenter PCF Format Reference*, which details PCF widgets and their attributes, from within Studio. Select **PCF Reference Guide** from the **Help** menu to access this guide in a separate browser window.

Guidewire ClaimCenter uses *page configuration format* (PCF) files to render the ClaimCenter interface. You use the Studio PCF editor to manage existing PCF files and create new ones. The PCF editor provides:

- Intelligent Gosu coding
- Instant feedback if a PCF file changes
- Drag-and-drop composition of PCF pages using graphical elements
- High-level view of PCF page groupings
- Ability to *localize* the display keys used in a PCF page

Using the Studio Page Configuration (PCF) editor, you can modify an existing PCF file or add a new PCF file and graphically build and manage its elements. Studio displays a list of all ClaimCenter PCF files as a tree structure in the Resources pane. To open a PCF file, select it from the resources tree.

Studio divides the PCF editor into three areas:

- The graphical *canvas* in the Studio center pane that provides a drag-and-drop capability for working with the PCF elements (widgets) on the PCF page.
- The specialized tabs in the Studio right-hand pane, which are:

Toolbox	Contains a search box and a list of elements (widgets) that you can insert into the page.
Structure	Shows the hierarchical structure of the PCF file as a tree, with the PCF elements being nodes on the tree.
Translations	Shows a list of all the display keys used in the PCF file and provides the ability to add locale-specific translations for each one.

- The **Properties** tabs at the bottom of the screen.

Any attempt to work within a PCF file opens a Studio dialog that prompts you to open the file for edit, if it is not already open.

The PCF Canvas

The *canvas* in the Studio center pane provides a drag-and-drop capability for working with the PCF elements (widgets) on the PCF page. On the canvas:

- Studio displays those elements that represent actual screen content (inputs, and similar items) as a simplified version of how they appear within ClaimCenter.
- Studio displays those elements that function primarily as containers (data views, for example) as light gray boxes, with a header indicating the element type and ID.
- Studio displays those 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, then opens the appropriate properties tab for editing whatever is providing the symbol. Finally, if necessary, Studio selects the symbol in the **Properties** tab.
- Studio displays those elements that are conditionally visible with a dotted border.
- Studio displays those 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.
- Studio displays RowIterator widgets with inferred header and footer cells in the position in which they appear within ClaimCenter.

Included Files

A widget that causes the contents of another file to be included displays the entire contents of that file inline, 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. If the included file has multiple modes, Studio displays a drop-down above the blue area with all the possible modes and displays the selected mode in the blue area. A Gosu expression defines the mode for an included section. This expression can be either a hard-coded string literal or it can evaluate a variable or a method call. A

hard-coded string guarantees that the included section always uses the same mode regardless of the data on the page. In this case, Studio shows only that mode and does *not* show a drop-down above the blue area.

IMPORTANT If you change the mode by using the drop-down, Studio does not prompt you to edit the file, nor does it actually make any change to the PCF file.

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.

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.

Creating a New PCF File

Guidewire organizes PCF files into a hierarchical structure. To create a new PCF file, you need to first decide its location in the PCF hierarchy. If the PCF structure does not currently contain a folder at the right level in the hierarchy to meet your needs, then you need to create one before proceeding.

To create a new PCF folder

1. Expand the **Page Configuration (PCF)** node in the Studio Resources tree.
2. Select a node one level above the level in which you need to create the new PCF folder (node).
3. Right-click and select **New → PCF folder**.
4. Enter the folder name in the **New Folder** dialog.

To create a new PCF file

1. Expand the **Page Configuration (PCF)** node in the Studio Resources tree.
2. Select the node in which you want to create the new PCF file.
3. Right-click and select **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.

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

Icon	File type	Icon	File type	Icon	File type	Icon	File type
	Entry Point		Menu Items		Screen		
	Exit Point		Menu Links Set		Tab Bar		
	Info Bar		Navigation Forward		Template Page		

The 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.

The 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.

The 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).

The Translations Tab

The **Translations** tab shows a list of all display keys used in the PCF file:

- If you click a display key in the list, Studio selects (outlines) the widget on the canvas that uses that display key.

- If you double-click a display key in the list, Studio opens an **Edit Display Key** dialog that shows the value of the display key in each locale defined within ClaimCenter. You can use this dialog to create locale-specific display keys that are specific to the context in which they exist.

See Also

- See “Localizing Display Keys” on page 483 for information on the various ways that you can localize display keys.

The 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
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.
Variables	If an element can have Variable children, Studio displays the Variables properties tab. This tab contains a child list of the Variable children for the selected element.
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.
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.
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.
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.
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.
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.
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.
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.
Toolbar Flags	If an element can have ToolbarFlag children, Studio displays the Toolbar Flags properties tab. This tab contains a child list of the ToolbarFlag children of the selected element.
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.

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.

IMPORTANT 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 on 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 on 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.

Note: 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.

Note: 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. Doing so 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, right-click, then click **Show element source**. 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.

A duplicate includes all attribute values and descendants. Studio selects the duplicate widget automatically.

Note: 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.

Note: Studio does not permit you to 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.

Note: Studio does not permit you to 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.
- Click **Paste** in the **Edit** commands on the menu bar.
- Click the **Paste** icon on the menu bar

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 349
- “Page Configuration Elements” on page 350
- “Getting Started Configuring Pages” on page 356

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. For an explanation of this hierarchy, consult “How ClaimCenter Interprets Modules” on page 89. *Guidewire expressly does not support editing PCF files outside of Guidewire Studio.*

Page Configuration Elements

This section discusses the following topics:

- Using Studio to Edit PCF Files
- What is a PCF Element?
- Types of PCF Elements
- Identifying PCF Elements in the User Interface

Using Studio to Edit PCF Files

This section provides instructions on using Guidewire Studio to edit PCF files. The

How Studio Interprets Modules

Within the `ClaimCenter/modules` directory, configuration resources are grouped in module folders, and they are evaluated at application startup according to a specific order:

Priority	Directory name	Description
1	<code>/configuration</code>	Edited versions of resources from any of the directories below.
2	<code>/cc</code>	Defines the base configuration for ClaimCenter.
3	<code>/platform</code>	Cross-product configuration including user interface, libraries, schema definition and other metadata.
4	<code>/core</code>	Core configuration resources supporting ClaimCenter.

These directories can contain distinct copies of the same resource file. In that case, the highest-priority copy would be in the first one found, causing any others to be disregarded.

For example, in the base install, the file `Desktop.pcf` resides in the `modules/cc/config/web/pcf/desktop` directory. If you edit this file from Studio, ClaimCenter creates a copy of the file and automatically places it in the `ClaimCenter/modules/configuration` directory. The original file remains, but ClaimCenter ignores it for as long as the edited copy exists. If you delete the edited copy, then ClaimCenter uses the original read-only copy again.

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.

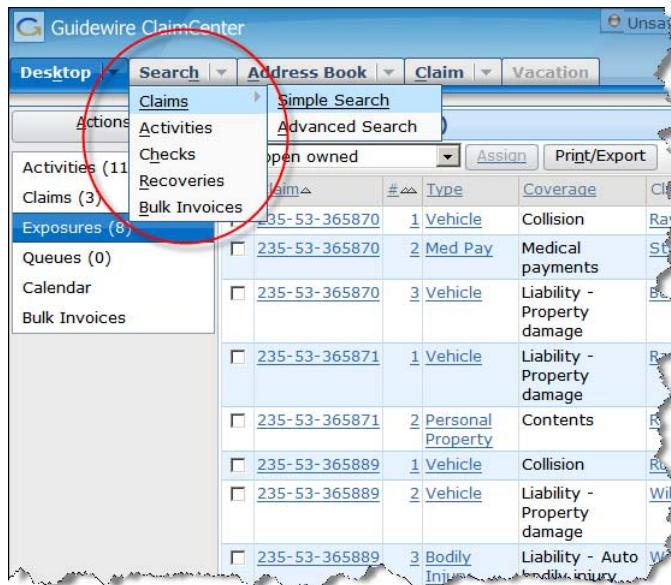
What is a PCF Element?

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.

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 PCF file `TabBar.pcf`. In Guidewire Studio, you can open `TabBar.pcf` in the PCF Editor. Clicking on the arrow next to the Search tab in this file causes the search menu items to appear:

Types of PCF Elements

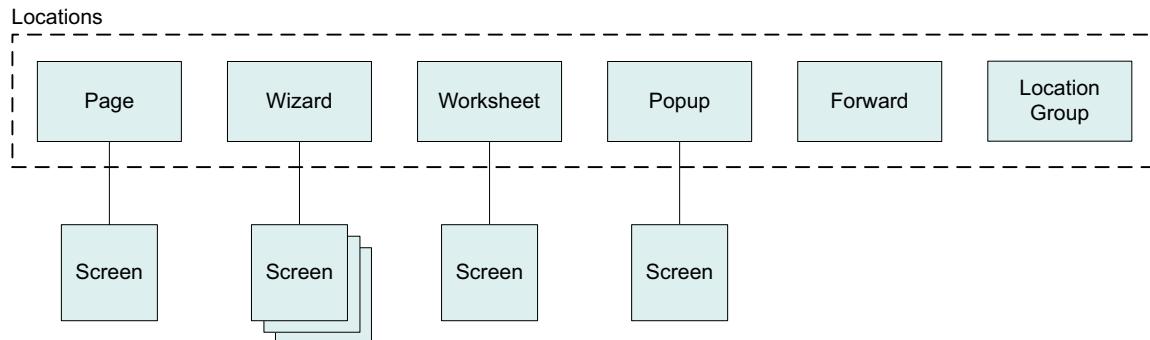
There are many kinds of PCF elements that you can define. These elements follow a hierarchical, container-based user interface model. To design them most effectively, you need understand the relationships between them thoroughly. Most PCF elements are of one of the following types:

- Locations
- Widgets

Locations

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

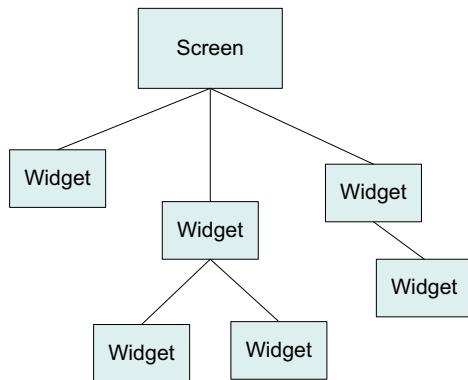
Location	Description
Page	A location with exactly one screen. The majority of locations defined in ClaimCenter are pages.
Wizard	A location with one or more screens, in which only one screen is active at a time. The contents of a wizard are usually not defined in PCF files, but are configured either in other configuration files or are defined internally by ClaimCenter.
Worksheet	A page that can be shown in the workspace, the bottom panel 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

#	Type	Coverage	Claimant	Adjuster	Status	Open	Remaining	Future	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	-	-

Parties Involved

Name	Role	Phone
Karen Egertson	Agent	213-457-6378
Brian Newton	Excluded Party	818-446-1206
Ray Newton	Insured	818-446-1206

Latest Notes (1 - 3 of 3)

Topic	General	Date	Author	Notes
Security		Jul 20, 2007 12:00 AM	Andy Applegate	Initial phone call with claimant
Type				Spoke with the claimant.

This diagram shows:

- The location is a page named `ClaimSummary`.
- The page contains a screen named `ClaimSummaryScreen`.
- The screen contains a “detail view” widget named `ClaimSummaryDV`.
- The screen contains multiple “list view” widgets, including `ClaimSummaryExposuresLV`, `PeopleInvolvedLV`, `NotesLV`, and so on.

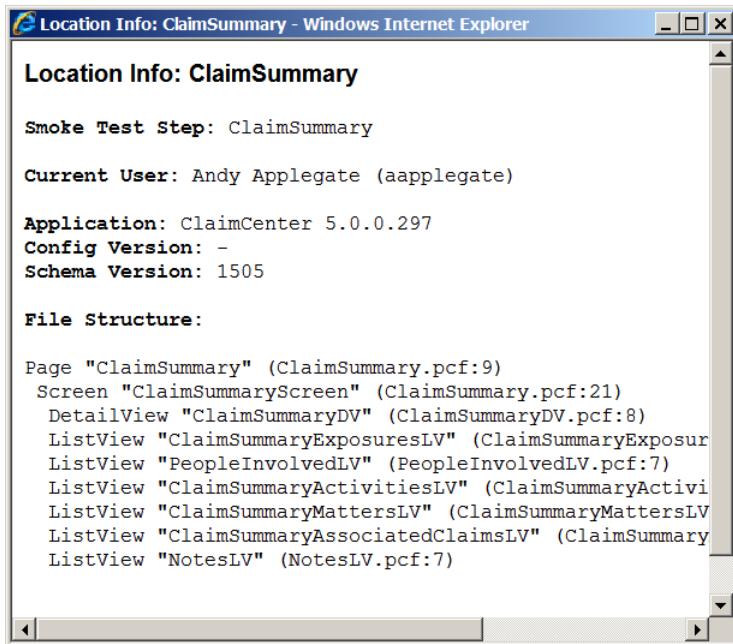
ClaimCenter provides the following tools that allows you to view the structure of any page and to see which PCF elements it uses:

- Location Info
- Widget Inspector

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 9.
- The page contains a screen named `ClaimSummaryScreen`, defined in the `ClaimSummary.pcf` file on line 21.
- 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

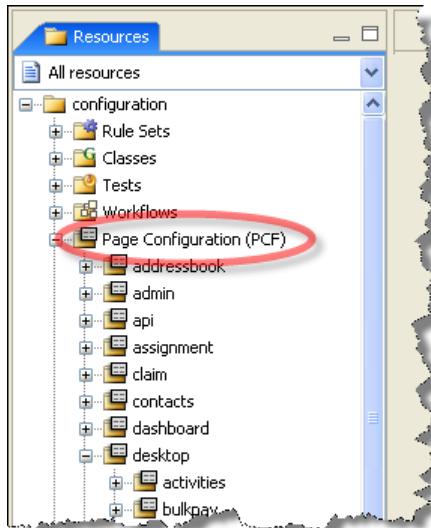
Finding an Existing Element To Edit

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

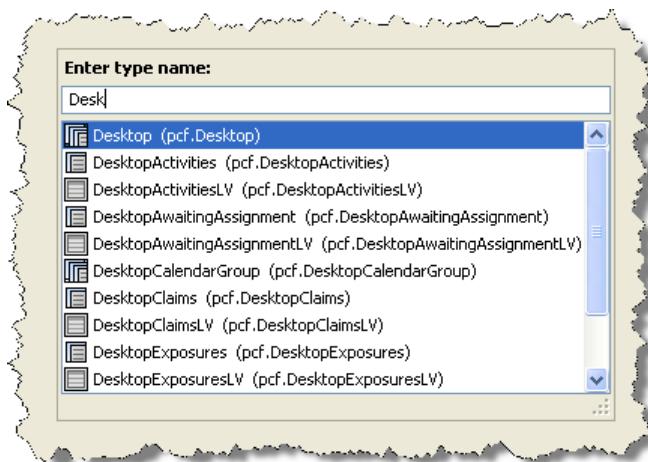
You can browse the PCF elements under the **Page Configuration** folder in Guidewire Studio:



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.



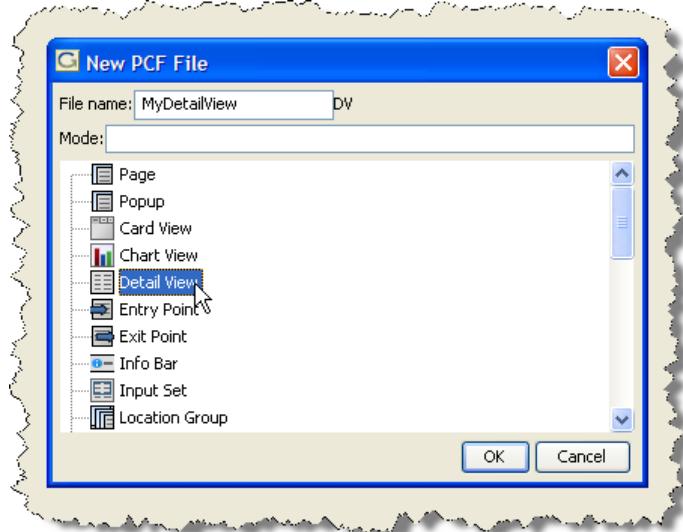
After you see the one you want, click on it and ClaimCenter opens the file in the PCF Editor.

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 resources tree, 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.

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 359
- “Detail View Panel” on page 359
- “List View Panel” on page 364

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:

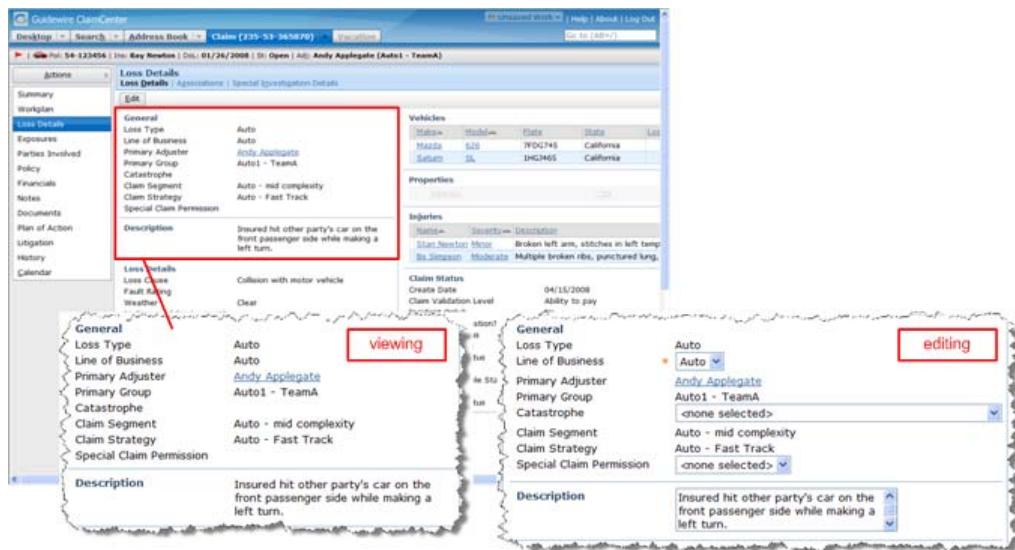
- **Detail View Panel**—A series of widgets laid out in one or more columns.
- **List View Panel**—A list of array objects, or any other data that can be laid out in tabular form.

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:

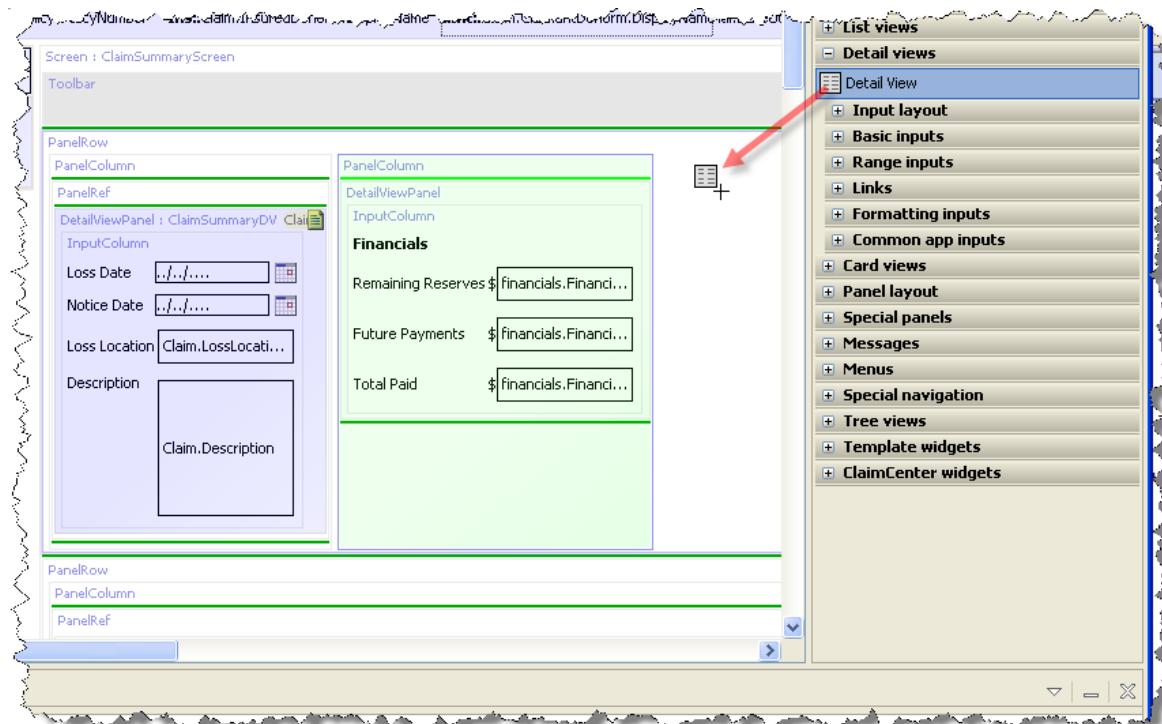


You can do the following:

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

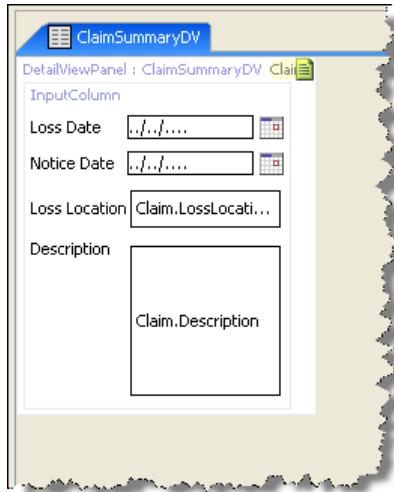
Define a Detail View

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:

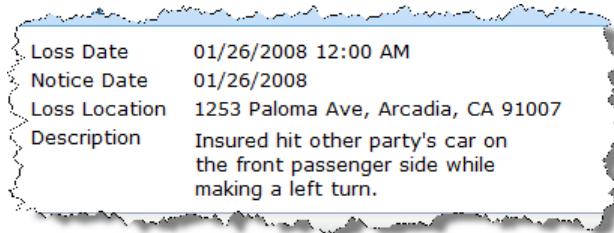


The `id` attribute is required; it identifies the panel so that it can be referenced by other PCF elements. The ID must be unique, and it must end with the text string `DV`.

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:

Report Request Summary

- Metro Report Type: Auto Accident
- Owner: Special User
- Date Sent: 04/16/2008
- Date Received: Received
- Status: Received

Investigating Agency Information

- Name of Investigating Agency:
- Type of Investigating Agency:
- Investigating Officer:
- Report Number:
- Officer Name:
- Phone:
- City of Investigating Agency: South Pasadena
- State of Investigating Agency: California

Column 1

Loss Details

- Coding Segment: Auto - low complexity
- Handling Strategy: Auto - Fast Track
- Jurisdiction State: California
- Other Carrier Involved: No
- Other Coverage: No
- Details:

Vehicle

- Owner: Ray Newton
- Owner of other vehicle:
- Primary Phone: 818-446-1206
- Address: 287 Kensington Rd. #1A, South Pasadena, CA 91105

Incident Overview

- Vehicle: 1992 Hyundai Elantra (HGM465 / California)
- Driver: Ray Newton
- Description: Minor damage
- Operator:
- Loss Estimate:

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:

InputColumn	
Exposure	
Loss Party *	<code>Exposure.LossParty</code>
Close Date	<code>Exposure.CloseDate</code>
Primary Coverage	<code>Exposure.PrimaryCoverage</code>
Resolution	<code>Exposure.ClosedOutcome</code>
Coverage Subtype	<code>Exposure.CoverageSubType</code>
Coding	
Segment	<code>Exposure.Segment</code>
Handling Strategy	<code>Exposure.Strategy</code>
Jurisdiction State	<code>Exposure.Jurisdiction</code>
Other Carrier Involvement	
Other Coverage	<code>Exposure.OtherCoverage</code>
Details	<code>ListviewTemplate</code>

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
Primary Coverage	Liability - Property damage
Coverage Subtype	Liability - Property Damage - Vehicle
Coverage	
Adjuster	Andy Applegate
Group	Auto1 - TeamA
Status	Open
Create Date	04/15/2008
Statistical Line	-
Validation Level	-
Claimant	Financials
Claimant	Ray Newton
Type	Owner of other vehicle
Other Carrier Involvement	
Other Coverage	No
Details	Insurer Policy # Contact
Financials	
Remaining Reserves	-
Future Payments	-
Total Paid	-
Total Recoveries	-
Net Total Incurred	-

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:

The diagram shows a configuration panel with the following structure:

General	
Loss Type	Auto
Line of Business	Auto
Primary Adjuster	Andy Applegate
Primary Group	Auto1 - TeamA
Catastrophe	
Claim Segment	Auto - mid complexity
Claim Strategy	Auto - Fast Track
Special Claim Permission	
Description	
Insured hit other party's car on the front passenger side while making a left turn.	
Loss Details	
Loss Cause	Collision with motor vehicle
Fault Rating	
Weather	Clear
In Course of Employment?	
Date of Loss	01/26/2008 12:00 AM

Annotations with red arrows:

- A red arrow labeled "Labels" points to the bolded section headers like "General", "Description", and "Loss Details".
- A red arrow labeled "Input dividers" points to the horizontal lines separating the "Description" section from the "Loss Details" section.

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:

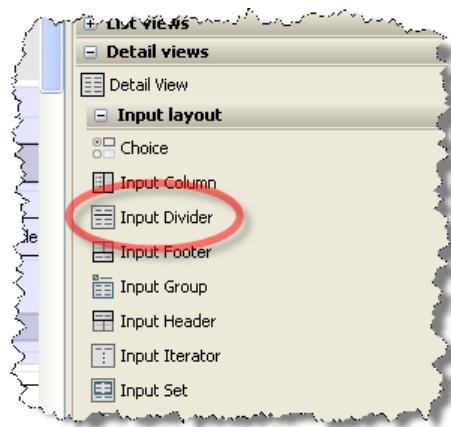


Set the `label` attribute to the display key to use for the label.

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—or 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.

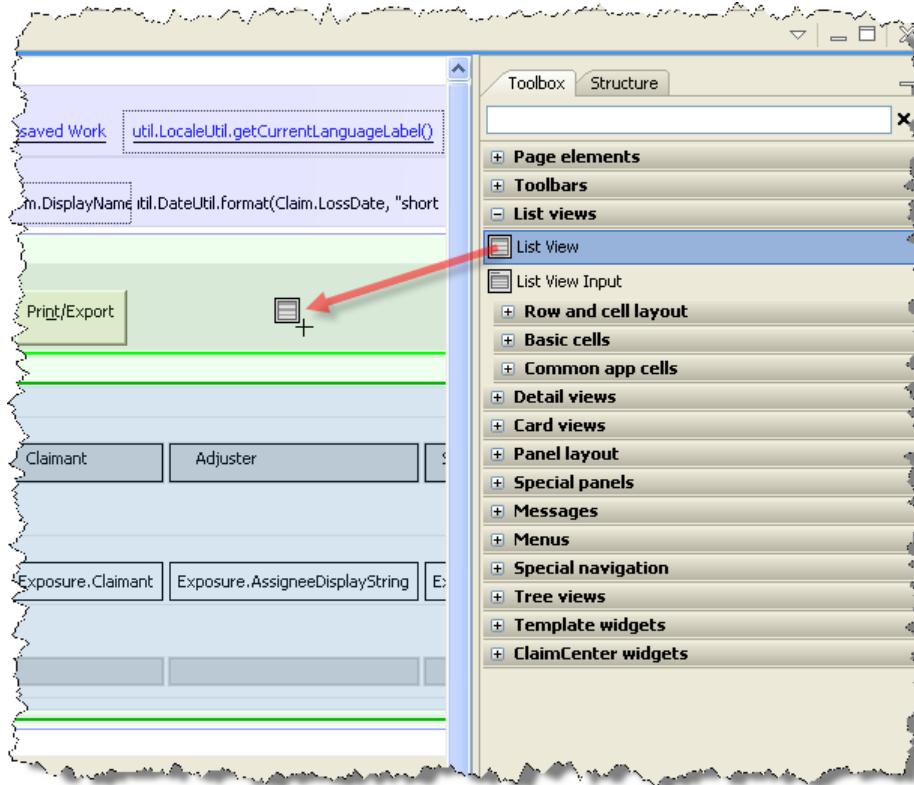
The following is an example of a list view:

The screenshot shows a 'List View' titled 'Exposures'. It includes a toolbar with buttons for 'Assign', 'Refresh', 'Close Exposure', 'Create Reserve', and 'Print/Export'. A 'filter' dropdown is set to 'All claimants'. The table has a 'header' row with columns: '#', 'Type', 'Coverage', 'Claimant', 'Adjuster', 'Status', 'Remaining Reserves', 'Future Payments', and 'Paid'. Below the header are four 'rows' of 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	-	-

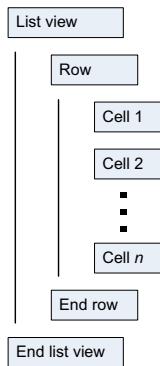
Define a List View

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:

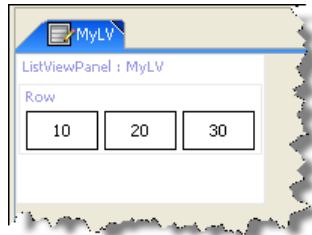


The `id` attribute is required; it identifies the panel so that it can be referenced by other PCF elements. The ID must be unique, and it must end with the text string `LV`.

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 367.

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:

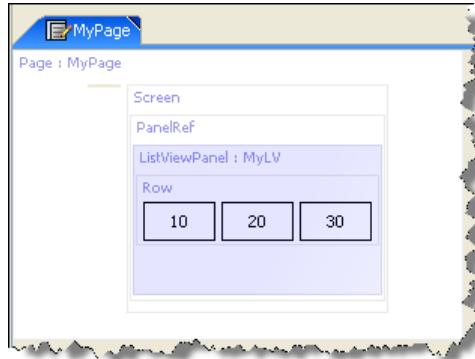
- Standalone
- Inline

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:

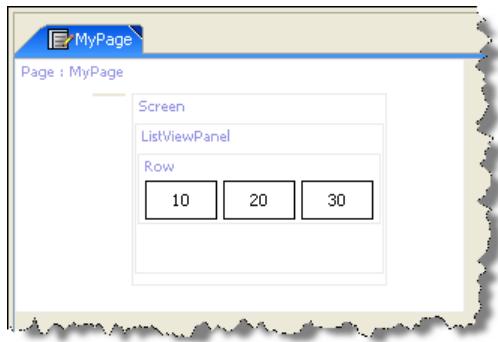


Set the `def` attribute of the `PanelRef` to the name of the list view; in this example, that is `MyLV`.

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.

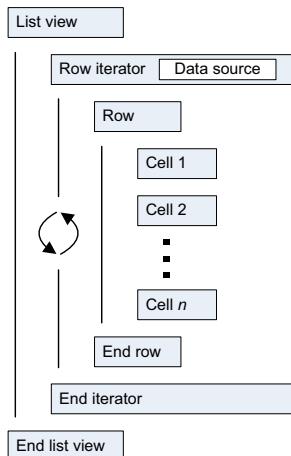
The following example defines an inline list view in a screen:



Iterate a List View Over a Data Set

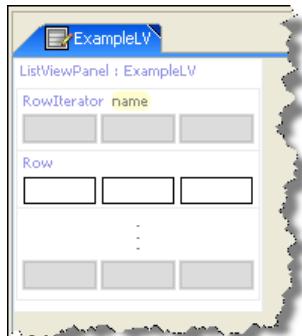
Most list views iterate over a data set and dynamically create a new row in the list for each record in the data set. The most common usage is showing an array of objects that belong to another object. For example, listing all activities that belong to a claim, or all users that belong to a group.

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.

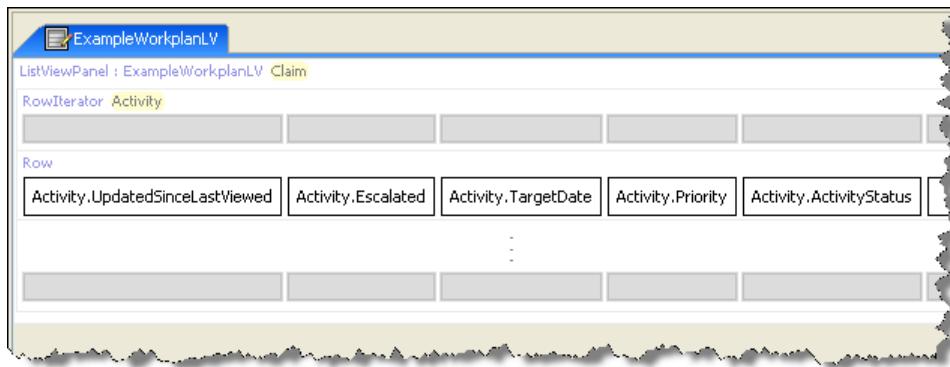
Define a row iterator with the **Row Iterator** PCF element. For example:



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 368.

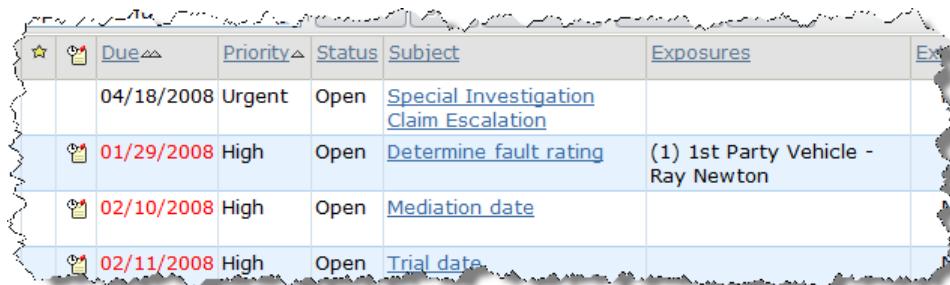
The `elementName` attribute is the variable name that represents the “current” row in the list. You can use this variable anywhere within the row iterator as the root object that refers to the current row.

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:



Choose the Data Source for a List View

Different uses of list views have different data requirements. Many list views are meant to display potentially long lists of items so that you can scan the list, pick an item, and go to a detail view of it. This is especially true for search results or lists of items (activities, claims, and so on) on the Desktop. For these kinds of list views, the items in the view are the result of a database query. (For example, this could be “all activities assigned to me that are due today or earlier”.) It is therefore important to make the data retrieval very fast and to be able to handle long lists. In other cases, however, the list is much simpler. For example, a list of exposures for a claim does not require a complex database search. The list is short, and the data is readily available as part of the overall claim. To accommodate these different needs, there are different kinds of data sources that you can use in a list view.

The data shown in a list view can come from the following sources:

Source	Description
Array	You can use any ClaimCenter array field as the data source for a list view. An array field is identified in the Data Dictionary 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	A query is a more complex data source. It is usually not just a full list of objects, but instead is a query that can involve filtering the returned data. A query can be one of the following:

Source	Description
QueryProcessor data field	A QueryProcessor field is identified in the Data Dictionary by the datatype QueryProcessor. 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, which loads both viewable and non-viewable notes, and then filtering the non-viewable ones out later.
Finder	A <i>finder</i> is similar to a QueryProcessor field, except that it is not exposed as a field in the data model. Instead, a finder is an internally-defined Java class that performs an efficient query. For example, the Activities page of the Desktop uses a list view based on the finder <code>Activity_finder.getActivityDesktopViewsAssignedToCurrentUser</code> . See “Find Expressions” on page 159 in the <i>Gosu Reference Guide</i> finders for more information.
Gosu find expression	You can define a custom Gosu method that returns the results of a <code>find</code> expression for use in a list view.

There are some differences in list view behavior depending on the kind of data source used. The following table summarizes the differences between a list view backed by an array and a list view backed by a query:

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:

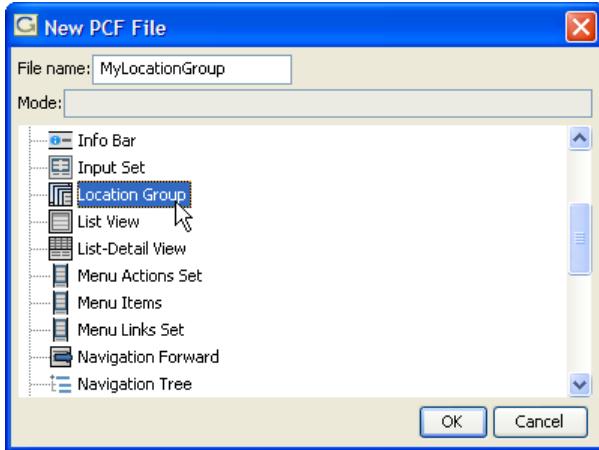
- “Location Group Overview” on page 371
- “Define a Location Group” on page 372
- “Location Groups as Navigation” on page 373

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. On the File menu, 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

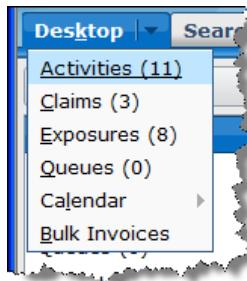
Depending on how a location group is used, ClaimCenter displays menus and other screen elements for navigating to the locations within that group. Any time that you navigate to a location group, ClaimCenter displays the first location within that group.

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

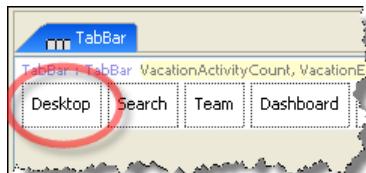
- Location Groups as Tab Menus
- Location Groups as Menu Links
- Location Groups as Screen Tabs

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:

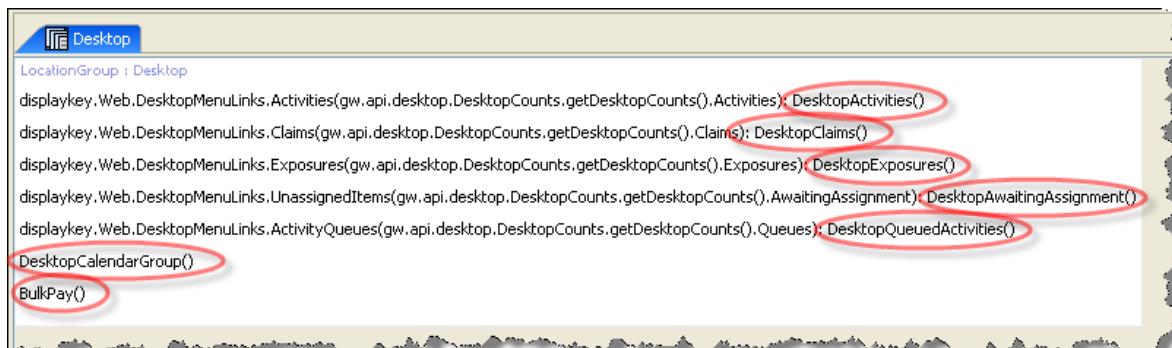


This tab is defined in the element named **TabBar** (under the **util** folder):



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.

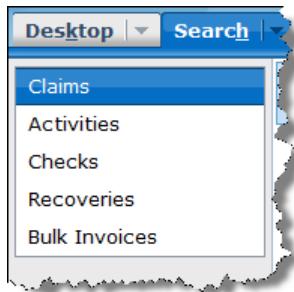
This location is a location group:



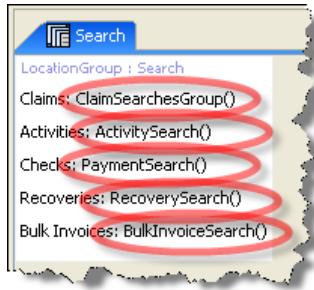
Inside this location group, there are multiple **Location Ref** elements defined, each one specifying a location. In this example, the locations referenced in the group correspond to the items in the **Desktop** menu. If the action for a tab is a location group containing more than one location, ClaimCenter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



The following is the definition of the Search location group:

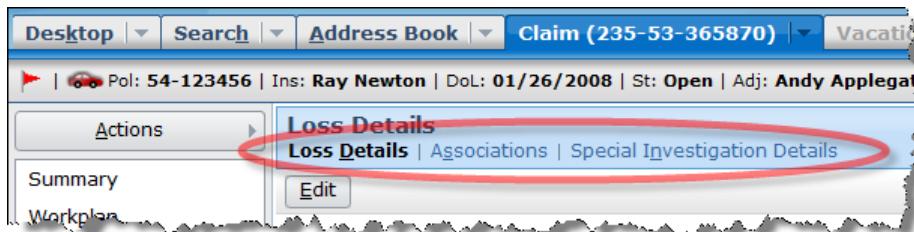


Inside this location group, there are multiple `Location Ref` elements defined, each one specifying a location.

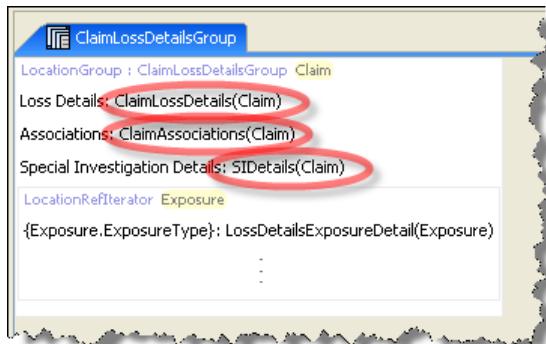
As ClaimCenter displays this location, it notices that it is a location group, and automatically creates menu links for each location within the group. Notice in this example that the `Location Ref` elements referenced in this group correspond to the items in the menu links.

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 `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 screen tabs 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 screen tabs.

Navigation

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

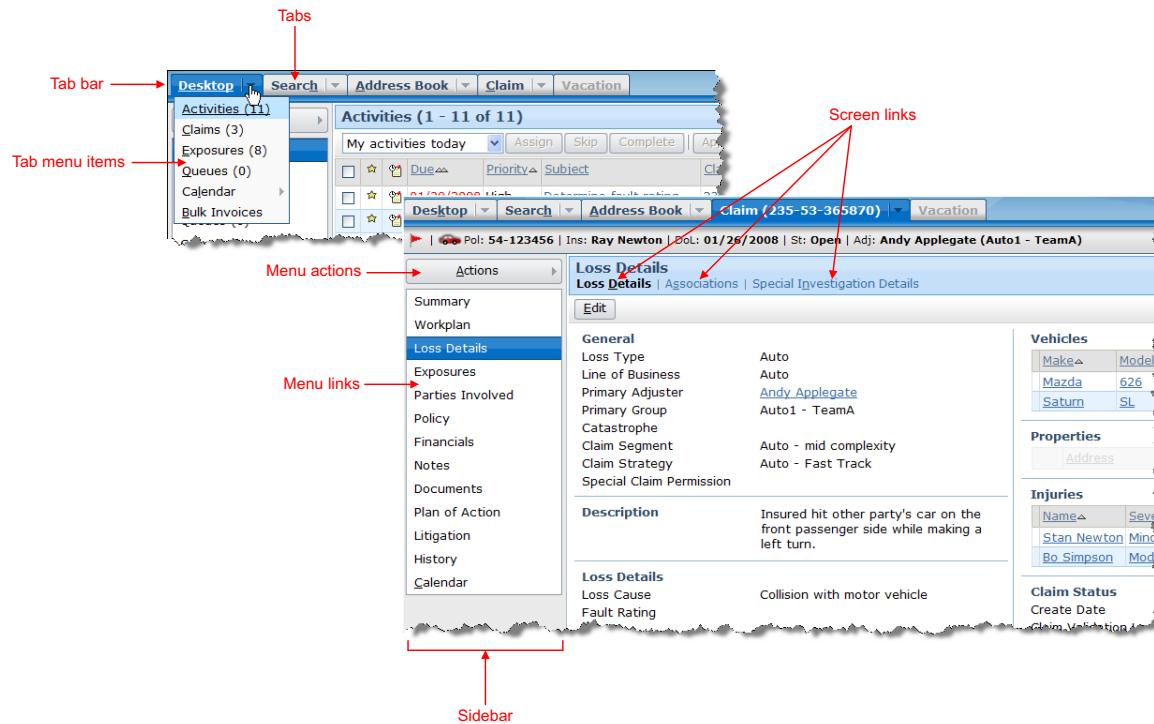
This topic includes:

- “Navigation Overview” on page 377
- “Tab Bars” on page 378
- “Tabs” on page 380

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:

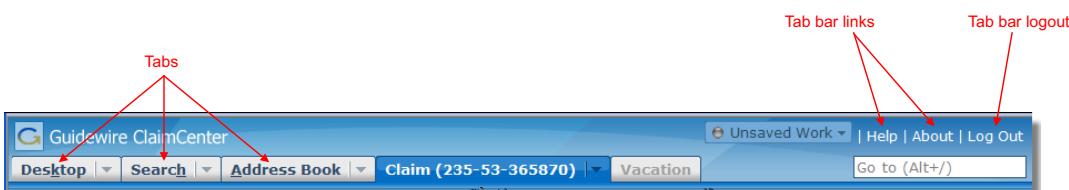


You can define the following types of navigation elements:

- **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.
- **Screen links**—Links within a screen that act in a similar way to tabs, allowing you to break a page into smaller sections.

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
- Define a Tab Bar Link
- Define the Logout Link

You can also configure the individual tabs on a tab bar. For more information, see “Tabs” on page 380.

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:



Define a Tab Bar Link

A tab bar link is a text link on the right side of the tab bar. A tab bar link can go to a web page or run JavaScript statements. Define a tab bar link inside a tab bar with the Tab Bar Link PCF element, as in the following example:



You can define as many tab bar links as you like by including the Tab Bar Link element multiple times. However, pay attention to how much room they occupy on the tab bar.

The action attribute defines what happens if you click on the link. The action string can be any Gosu statement, or it can begin with one of the following prefixes:

- **http:**—Go to the following URL. For example: action="http://www.guidewire.com"

- **https:**—Same as http: but using the https protocol instead.
- **javascript:**—Run the following JavaScript. For example: action="javascript:alert('Hello');"

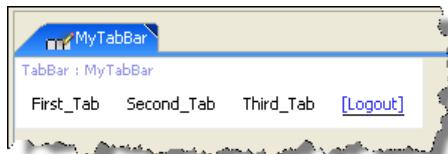
You can open a URL in a new window with the following action:

```
javascript:void window.open('http://www.guidewire.com');
```

The void prevents the main ClaimCenter window from being redirected.

Define the Logout Link

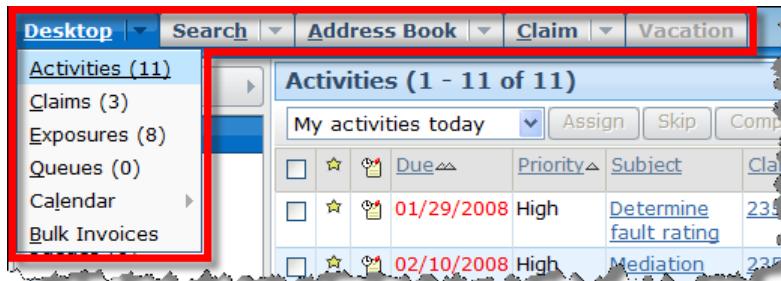
The tab bar logout is a special link on the right side of the tab bar that logs the current user out of ClaimCenter. It also warns the user of any unsaved work. Define a tab bar logout inside a tab bar with the <TabBarLogout> PCF element, as in the following example:



The action attribute must begin with the prefix `logout:` and then specify the entry point to go to after logging out.

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

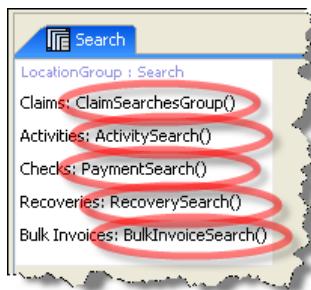
Define a Tab Menu From a Location Group

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 `MenuItem` 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 383
- “How to Configure Spell Check” on page 384

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:

- Any time 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 any time that a user performs an action that initiates a spell check. Depending on how you configure your application, the functionality can become active by simply leaving an edited field (the field loses focus) or by pressing 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 check utility to correct any spelling errors.

If you configure the **Check Spelling** button, pressing the button causes ClaimCenter to check all fields in the current page that you configure for spell checking. This is 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 you configure the **Subject** field on a note for spell checking. A user can select the field and edit it. If the user then leaves the field, ClaimCenter calls the spell 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 check utility. You are responsible for providing a dictionary or otherwise ensuring the ability of your spell check utility to determine what is—and is not—a misspelling.

WARNING ActiveX controls have known security vulnerabilities. Contact Guidewire support for more information if you plan to use an ActiveX control in production.

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 Resources tree under `Web Resource → web/templates → util`.) File `SpellCheckFrame.html` contains JavaScript that ClaimCenter uses to call your spell check utility. Guidewire also implements a very simple test spell checker utility within the JavaScript of 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 spelling check 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:

- | | |
|------------------------------------|---|
| <code>CheckSpellingOnChange</code> | Controls whether ClaimCenter does field-by-field checking. If set to <code>true</code> , spell check runs immediately each time the cursor leaves a checkable text field or text area field. The default value is <code>false</code> . |
| <code>CheckSpellingOnDemand</code> | Controls whether the Check Spelling button appears in the ClaimCenter interface. If set to <code>true</code> , 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 <code>false</code> . |

IMPORTANT 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 key entities. You can customize the **Search** tab to add new search criteria or modify or remove existing criteria. It also describes how the search feature integrates with the underlying data model and the XML and PCF 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 contacts, 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 introducing them into a production database.

This topic includes:

- “The ClaimCenter Search Functionality” on page 387
- “File search-config.xml” on page 388
- “The Criteria Definition Element” on page 389
- “The Criterion Subelement” on page 390
- “The Criterion Choice Subelement” on page 392
- “The Array Criterion Subelement” on page 394
- “Deploying Customized Search Files” on page 395
- “Steps in Customizing a Search” on page 396

The ClaimCenter 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. In Studio, these files are located under **PCF → search**. Each search detail view references two subviews, each contained in 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.

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**.

IMPORTANT Search criteria entities are *virtual* entities, which have no underlying table in the ClaimCenter database. Rather, they are non-persistent and only exist in the session in which you use them.

File search-config.xml

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 <code>CriteriaDef</code> elements to <code>search-config.xml</code> . Instead, simply modify the contents of any existing <code>CriteriaDef</code> elements. The ClaimCenter search code uses these criteria definitions as it builds a query. If you add a new <code>CriteriaDef</code> 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. This means that the search criteria column must match against 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 in a criterion choice.
ArrayCriterion		Specifies that the search query uses a simple join against an array entity.

The Criteria Definition Element

A `<CriteriaDef>` element specifies the mapping from a search criteria entity to the target entity on which to search. For example, a `<CriteriaDef>` element can specify a mapping between a `DocumentSearchCriteria` entity and a `Document` entity. A `<CriteriaDef>` element uses the following syntax:

```
<CriteriaDef entity="entityName" targetEntity="targetEntityName">
```

These attributes have the following definitions:

<code><CriteriaDef></code> attributes	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, in ClaimCenter, 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 in the search query itself.

WARNING 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. Guidewire **strongly** recommends you do not remove `<CriteriaDef>` elements that exist in the base configuration.

A <CriteriaDef> element can have the following subelements:

<CriteriaDef> subelements	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 defined 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
PaymentSearchCriteria	Check CheckRpt
RecoverySearchCriteria	Recovery
UserSearchCriteria	Attribute AttributeUser AuthorityLimitProfile User

The Criterion Subelement

In 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:

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.

Attribute	Required	Description
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 <code>true</code>, then the Criterion uses an <code>eq</code> (equality) match. If this attribute evaluates to <code>false</code>, then the Criterion uses the <code>matchType</code> that the Criterion specifies to perform the match. <p>For example:</p> <pre><Criterion property="StringProperty" forceEqMatchType="FlagProperty" matchType="startsWith"/></pre> <p>This code uses a <code>startsWith</code> match for <code>StringProperty</code> unless the <code>FlagProperty</code> on the criteria entity is <code>true</code>, in which case, the match uses an <code>eq</code> match type.</p>
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:

Match type	Evaluates to	Use with data type	Comments
<code>contains</code>		<code>String</code>	Guidewire recommends that you avoid using the <code>contains</code> match type, if at all possible. This match type is the most expensive type in terms of performance impacts.
<code>eq</code>	<code>equals</code>	Numeric or Date	
<code>ge</code>	<code>greater than or equal</code>	Numeric or Date	
<code>gt</code>	<code>greater than</code>	Numeric or Date	
<code>le</code>	<code>less than or equal</code>	Numeric or Date	
<code>lt</code>	<code>less than</code>	Numeric or Date	
<code>startsWith</code>		<code>String</code>	The <code>startsWith</code> match type is also very expensive in terms of performance. Use it with caution.

Performance. 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. Consult a database expert.

IMPORTANT Guidewire **strongly** recommends that you avoid the `contains` match type if at all possible. It is the most expensive type of match (followed by `startsWith`).

IMPORTANT Guidewire strongly recommends that you ensure that appropriate indexes are in place if you change the search criteria. For example, suppose that you add a column that is the most restrictive equality condition in your search implementation. In this case, you need to consider adding an index with this column as the leading key column also.

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 attempt to change them.* **Guidewire expressly prohibits** the addition of new required fields in the ClaimCenter search screens. Adding your own required search criteria can cause performance issues severe enough to bring down the production database.

Do not even attempt to change the `match type` of the existing required properties. **Guidewire expressly prohibits** you from doing this due to restrictions on configuring fields on tables that are joined to the search table.

The Criterion Choice 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:

<code><CriterionChoice></code> attributes	Required	Description
property	Yes	One of the following: <ul style="list-style-type: none">• <code>DateCriterionChoice</code>—Use to specify the options available to restrict a date search.• <code>ArchiveDateCriterionChoice</code>—Use to specify the options available to restrict a date search on an archived object.• <code>FinancialCriterion</code>—Use to specify the options available to restrict a financial field search.
init	No	Gosu string that <code>DateCriterionChoice</code> and <code>ArchiveDateCriterionChoice</code> use to initialize the criterion choice.

A `<CriterionChoice>` element can have the following subelement:

<code><CriterionChoice></code> subelements	Description
Option	Describes a single choice in 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:

<code><Option></code> attributes	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 This attribute is optional but Guidewire requires that you specify this attribute if you add a new <code><Option></code> 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 = &quot;claim&quot;;
    DateCriterionChoice.DateSearchType = &quot;fromlist&quot;;
    DateCriterionChoice.DateRangeChoice = &quot;n365&quot;;
    DateCriterionChoice.ChosenOption = &quot;Java.Criterion.Option.Claim.LossDate&quot;">
    <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 <i>default</i> 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

Notes

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 in 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>
```

Setting the Property Attribute to ArchiveDateCriterionChoice

The `<ArchiveDateCriterionChoice>` element is similar to the `<DateCriterionChoice>` element and contains the same property and `init` attributes. However, you use this element in searches of archived objects only.

The following example (from Guidewire ClaimCenter) illustrates the use of the `<ArchiveDateCriterionChoice>` attribute on the `<CriterionChoice>` subelement configured on the `ClaimInfo` object.

```
<CriterionChoice property="ArchiveDateCriterionChoice"
  init="DateCriterionChoice.SearchType = "claimInfo";
  DateCriterionChoice.DateSearchType = "fromlist";
  DateCriterionChoice.DateRangeChoice = "n365";
  DateCriterionChoice.ChosenOption = null;>
  <Option label="Java.Criterion.Option.ArchiveClaim.LossDate" targetProperty="LossDate"/>
</CriterionChoice>
```

The Array Criterion 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> attributes	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 398.

Notes

- 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.
- 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 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>
```

- It is not possible to specify a `matchType` property on this criterion. Guidewire supports equality matching only for this type of criterion.

Deploying Customized Search Files

Guidewire (as always) recommends that you first make your search customization file changes in your development environment. You must then must 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 it 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 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.*

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 attempt to change them. Guidewire expressly prohibits the addition of new required fields in the ClaimCenter search screens. Adding your own required search criteria can cause performance issues severe enough to bring down the production database.

IMPORTANT Do not attempt to change even the match type of the existing required properties. Guidewire expressly prohibits you from making this change due to restrictions on configuring fields on tables that are joined to the search table.

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 illegal, it has no effect as ClaimCenter ignores the change.

The following examples illustrate customizing the ClaimCenter application by adding additional search fields:

- Adding an Optional Search Field
- Adding an Optional Array Search Field

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.

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.

See Also

- “The ClaimCenter Search Functionality” on page 387
- “File search-config.xml” on page 388
- “The Criteria Definition Element” on page 389

To add an optional search field

1. Create file `Claim.etx` in `Data Model Extensions → extensions` folder, if one does not already 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 238 for details of this process, if necessary.

2. Create file `ClaimSearchCriteria.etc` in **Data Model Extensions** → **extensions** folder, 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.

3. 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.

4. In 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">
    ...

```

5. Create the `PercentComplete` display key using the **Display Keys** editor. Enter the following:

Display Key Name	JSP.ClaimSearch.Claims.PercentComplete
Default Value	Percent Complete

See “Using the Display Keys Editor” on page 175 for details if necessary.

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. Open **Page Configuration (PCF)** → **search** → **claims** → **ClaimSearchOptionalInputSet** 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	Completion
label	displaykey.JSP.ClaimSearch.Claims.PercentComplete
required	false
value	ClaimSearchCriteria.PercentComplete

See “Using the PCF Editor” on page 337 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 “Commands Reference” on page 69 in the *Installation Guide*. Specifically, see the Build Tools topic.

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.

See Also

- “The ClaimCenter Search Functionality” on page 387
- “File `search-config.xml`” on page 388
- “The Criteria Definition Element” on page 389

To add an optional array search field

1. Create file `ClaimCode.eti` in **Data Model Extensions → extensions** folder. 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.etcx` in **Data Model Extensions** → `extensions` folder, 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 238 for details of this process, if necessary.

3. Create file `ClaimSearchCriteria.etcx` in **Data Model Extensions** → `extensions` folder, 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. In 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. Create the `ClaimCode` display key using the **Display Keys** editor. Enter the following:

Display Key Name	JSP.ClaimSearch.Claims.ClaimCode
Default Value	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. Open **Page Configuration (PCF)** → **search** → **claims** → **ClaimSearchOptionalInputSet** 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 337 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 “Commands Reference” on page 69 in the *Installation Guide*. Specifically, see the Build Tools topic.

11. Restart the application server.
12. Log into Guidewire 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.

Configuring Special Page Functions

This topic describes how to configure special functionality related to pages.

This topic includes:

- “Adding Print Capabilities” on page 401
- “Linking to a Specific Page: Using an EntryPoint PCF” on page 407
- “Linking to a Specific Page: Using an ExitPoint PCF” on page 410
- “Populating a PCF Template: Using a TemplatePage PCF” on page 412

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 404 for details.

Note: All client machines must have a supported version of the Acrobat Reader available to support printing. See the *ClaimCenter Installation Guide* for the version required for this release.

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 35.

<code>DefaultContentDispositionMode</code>	Specifies the Content-Disposition setting to use if the content to be printed is returned to the browser. Must be either “attachment” (the default) or “inline”.
<code>PrintFontFamilyName</code>	Sets the name of font family to use for output. The default is “sans-serif”.
<code>PrintFontSize</code>	Sets the page font size. The default is 10 points.
<code>PrintFOPUserConfigFile</code>	(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`. Refer to the *Gosu Reference Guide* for information about how to access and use API methods in ClaimCenter.

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(GroupInfo :
    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
<code>PrintDetail</code>	Instructions on how to print elements if you elect to print details.
<code>PrintGroup</code>	Defines a set of pages to print. This element contains one or more <code>PrintSection</code> elements. ClaimCenter represents each <code>PrintSection</code> element in the interface with a radio button. You can customize a <code>PrintGroup</code> . See "Customizable Printing" on page 403 for more information on using a customizable group.
<code>PrintLocation</code>	Specifies a specific location to print. This element takes one or more <code>PrintLocationDetail</code> elements.

Element	Description
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 or more PrintOptionLocation subelements.
PrintOptionLocation	Specifies a page (location) to print any time that you select a PrintGroup. This element can contain one PrintDetail.
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. (This means that 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"
            filters="FilterSet.AllFilter"/>
    </PrintOption>
    <PrintOption id="HistoryAssignmentOption" label="displaykey.Java.HistoryFilter.Assignment">
        <PrintOptionLocation locationRef="ClaimHistory(Claim)" listViewRef="HistoryLV"
            filters="FilterSet.AssignmentFilter"/>
    </PrintOption>
</PrintSection>
```

```
        filter="FilterSet.AssignmentFilter"/>
    </PrintOption>
    <PrintOption id="HistoryViewingOption" label="displaykey.Java.HistoryFilter.Viewing">
        <PrintOptionLocation locationRef="ClaimHistory(Claim)" listViewRef="HistoryLV"
            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:

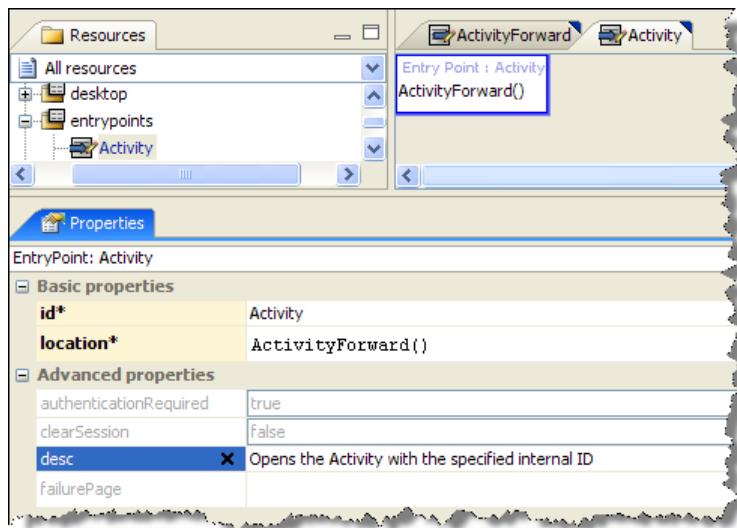
Resources → Page Configuration (PCF) → entrypoints

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 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.

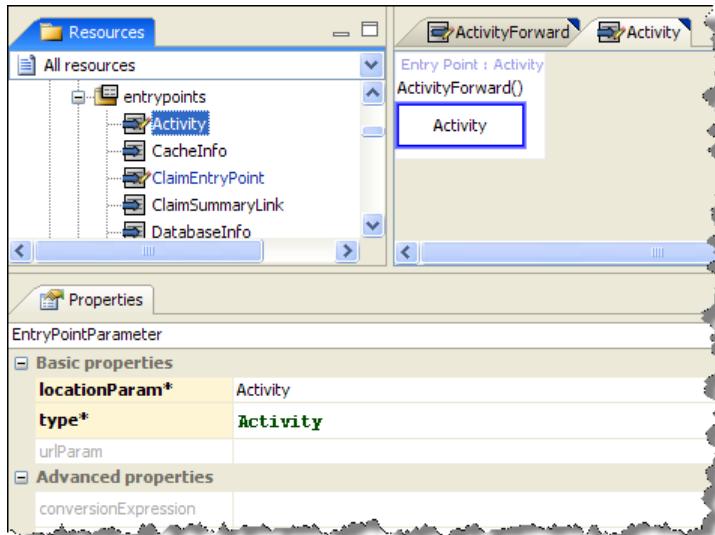
To implement this functionality, you must create an EntryPoint PCF (in the PCF → entrypoints folder). The following graphic illustrates an EntryPoint PCF.



The EntryPoint PCF contains the following parameters:

authenticationRequired	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.
clearSession	If true, clears the server session for this user as the user enters this entry point
desc	Currently, does nothing.
failurePage	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.
id	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 http://myserver/myapp/XXX.do .
location	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 409 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 LocationParameter on the <code>EntryPoint</code> target location that this parameter sets.
<code>optional</code>	Specifies whether the parameter is optional. If set to true, 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 <code>Activity</code> 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*, re-direct 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:

Resources → 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 **Page Configuration (PCF)** → **exitpoints** and 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.

Populating a PCF Template: Using a *TemplatePage* PCF

You can configure PCF template pages that retrieve data from the ClaimCenter database. Using template pages, you can develop an HTTP client program that requests data from ClaimCenter and returns it as formatted data. Typically, a template page includes Gosu that executes with the scope of the page. The result is that any parameters or variables declared within a page are available within the template.

A Simple Template Page

A **TemplatePage** element encloses each template page. By default, template pages return HTML. However, you can set the `contentType` attribute on the **TemplatePage** element to any valid MIME type such as text or XML. Refer to the following web site for a list of valid MIME types:

<http://www.iana.org/assignments/media-types/>

Creating a Template PCF Page

To create a simple template page in Studio, perform the following steps:

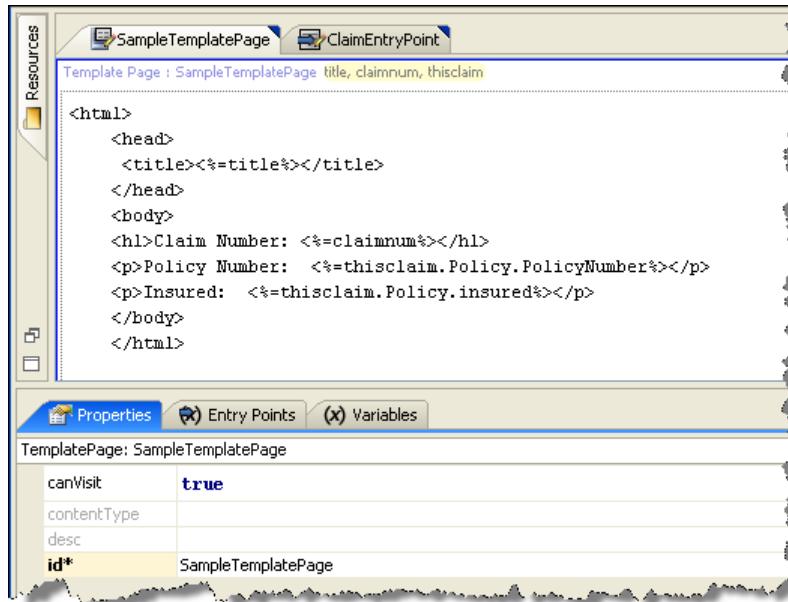
1. Navigate to the PCF folder in which you want to create the template PCF page.
2. Right-click and select **New** → **PCF File**.
3. Do the following in the **New PCF File** dialog:
 - Enter the name of the PCF.
 - Set the file type to **Template Page**.Studio creates your PCF template page and colors it red. This is because the PCF page is not yet valid.
4. Select **Template** from the list of **Template Widgets** and drag it onto the existing **TemplatePage** area on the screen. Studio inserts a new square outlined in blue into the template page.
5. Enter the template text (generally an HTML page) in the **value** field in the **Properties** tab at the bottom of the screen.

There are several issues in working with template pages in Studio of which you need to be aware:

- Studio does not display the **Properties** tab for the template widget if you click within the large open template area. Instead, you must select the outline of the template widget. This means if you click away from the template widget, you must select its outline to return to its **Properties** tab.
- Studio does not render the text in the **value** field until you actively save the PCF page. You can also navigate away from the page, in which case, Studio automatically saves the currently open view. This means you do not see anything in the template area on the screen, until you (or Studio) saves your work.
- Studio does not expand the **value** field to display all of your entered text until you save your work.

Creating a Template Page: An Example

Suppose that you create the following simple template page (within ClaimCenter Studio):

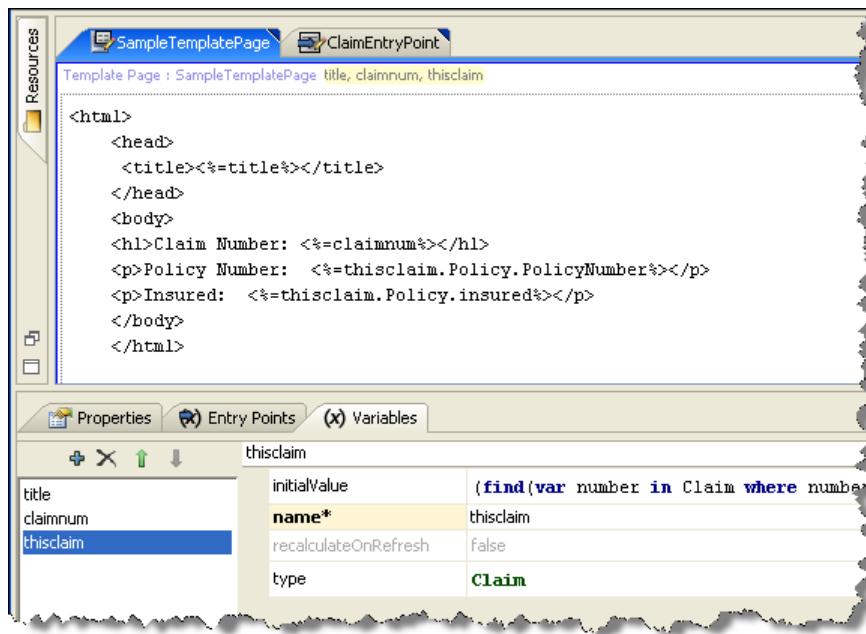


This is a simple template page that defines a `title`, `claimnum`, and a `thisclaim` variable for use in creating a simple display of claim information. It contains the following HTML code with the embedded variables:

```
<html>
  <head>
    <title><%=title%></title>
  </head>
  <body>
    <h1>Claim Number: <%=claimnum%></h1>
    <p>Policy Number: <%=thisclaim.Policy.PolicyNumber%></p>
    <p>Insured: <%=thisclaim.Policy.insured%></p>
  </body>
</html>
```

When the ClaimCenter server executes the page, it uses these variables to build the page contents. Template pages are PCF pages and, by convention, Guidewire recommends that you store templates in the `PCF → templates` folder. As PCF pages, the XML parser validates each template page.

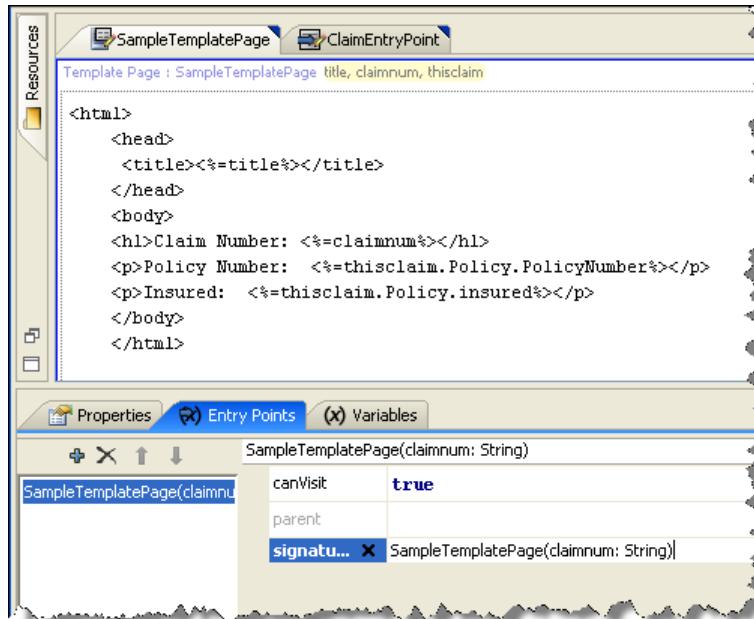
You also need to set the variables used by this template page:



The following list describes these variables.

Name	Type	Initial value
title	String	"Sample Title"
claimnum	String	"claimnum"
thisclaim	Claim	(find(var number in Claim where number.ClaimNumber == claimnum).getAtMostOneRow())

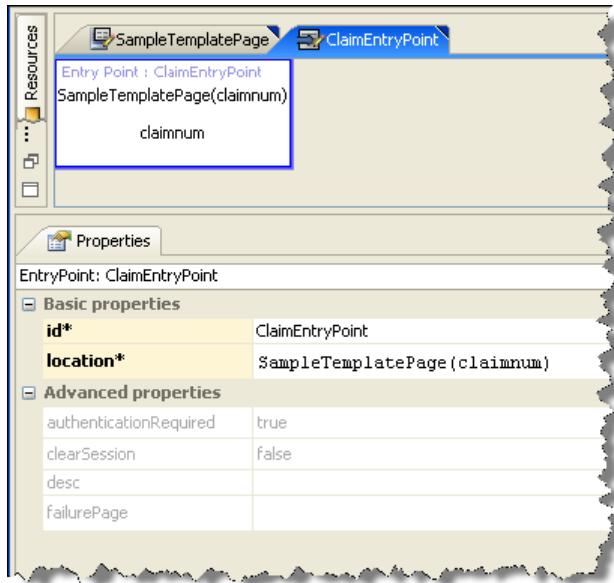
The final item that you must define for SampleTemplatePage is the page signature to be used by a calling page. For example, suppose that you simply want to pass in the claim number and have ClaimCenter use it to determine the other variables. In that case, enter the following in the **EntryPoints** tab of SampleTemplatePage.



Creating an Entry Point for a Template Page

Template pages are outside the standard ClaimCenter interface. For this reason, you must define an entry point for the page so that a user can access the page from URL.

To add an entry point for this page, create an **EntryPoint** file and place it in the **PCF → entrypoints** folder. For example, suppose you create the following **ClaimEntryPoint**:



Notice that the **location** property specifies the ID of the template page to go to upon entering this ClaimCenter entry point. The **location** value must match exactly what you specified as the **signature** on the **EntryPoints** tab of **SampleTemplatePage**. In both cases, it reads:

SampleTemplatePage(claimnum)

Note: After you establish the entry point, you must push the PCF files to the development server or rebuild and redeploy your configuration to a production server.

Passing Parameters to the Template

Entry points can take parameters and these parameters can be passed into and used by the template page. To call this page, enter the following URL into your browser. (The ClaimCenter server must be running for this to work).

```
http://server:port/cc/  
SampleTemplatePage.do?loginName=aapplegate&loginPassword=gw&claimnum=235-53-365870
```

ClaimCenter builds the HTML body by referencing the claim located with the passed-in claim number. It then displays the result as the following browser page.



The entry point requires authentication as property `authenticationRequired` was set to `true` in `ClaimEntryPoint`. To allow users to bypass the ClaimCenter login, the URL specifies the `loginName` and `loginPassword` parameters for a valid user.

You must also enter a valid claim number as `SampleTemplatePage` expects to receive a claim number from `ClaimEntryPoint`.

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 419
- “Workflow in Guidewire Studio” on page 419
- “Understanding Workflow Steps” on page 421
- “Using the Workflow Right-Click Menu” on page 422
- “Using Search with Workflow” on page 422

Workflow in Guidewire ClaimCenter

Note: For information on workflow structure and design, see “Guidewire Workflow” on page 423.

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`

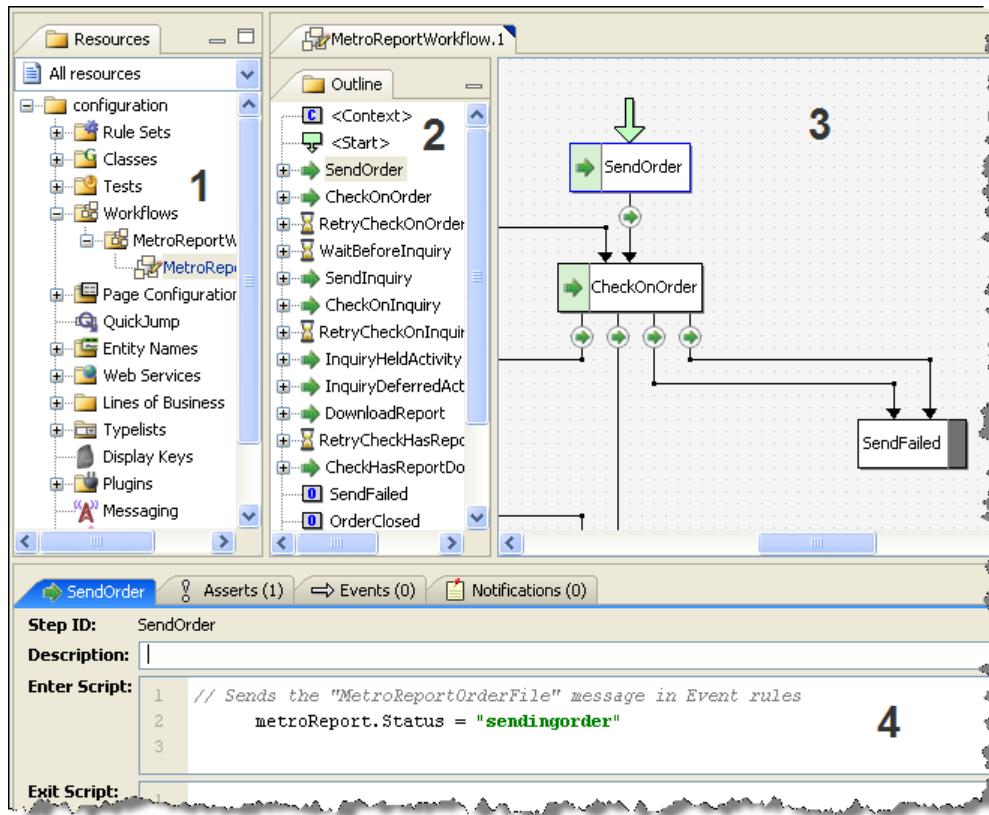
Workflow in Guidewire Studio

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, select **Workflows** from the Studio Resources tree, 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 incremented 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 429

Element	Editor	Description	See
<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 429
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 431
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 432
ActivityStep		An ActivityStep is similar to 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.	"ActivityStep" on page 433
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 434
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 437
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 438
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 439
Outcome		Indicates a possible outcome for the workflow. This step is special. It indicates that it is a <i>last</i> step, out of which no branch leaves.	"Outcome" on page 435
<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 429

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 428.

Type	Workflow contains	Icon	Step	Description
AutoStep	Zero, one, or more		Step1	Step that ClaimCenter guarantees to finish immediately
ManualStep	Zero, one, or more		Step2	Step that waits for an external TRIGGER to occur or a TIMEOUT to pass
ActivityStep	Zero, one, or more		Step3	Step that waits for one or more activities to complete before continuing
MessageStep	Zero, one, or more		Step4	Special-purpose step designed to support messaging-based integrations
Outcome	One or more		Outcome	Special final step that has no branches leading out of it

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.

Note: You can localize a workflow step name as well. See “Localize a Workflow” on page 497 for details.

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 Search menu. 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.

Studio opens a search panel 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.

Note: You can search on a localized text string as well.

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 424
- “Workflow Structural Elements” on page 428
- “Common Step Elements” on page 429
- “Basic Workflow Steps” on page 431
- “Step Branches” on page 436
- “Creating New Workflows” on page 440
- “Instantiating a Workflow” on page 444
- “The Workflow Engine” on page 446
- “Workflow Subflows” on page 450
- “Workflow Administration” on page 450
- “Workflow Debugging and Logging” on page 452

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.)

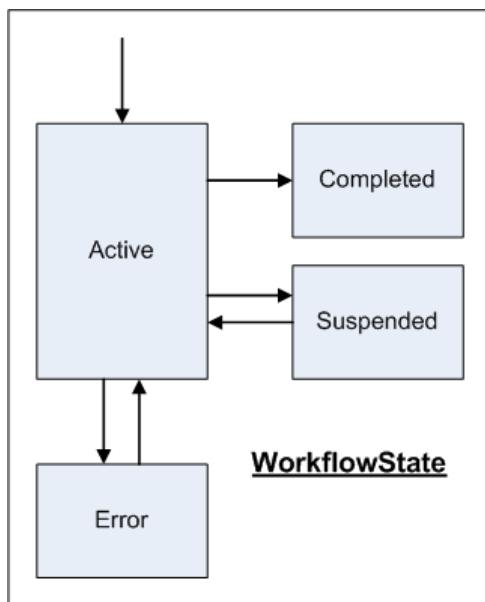
The newly created instance takes the form of a database entity called `Workflow`. (For more information on the `Workflow` entity, consult the ClaimCenter *Data Dictionary*.) Because ClaimCenter creates the `Workflow` entity in a bundle with other changes to its associated business data, ClaimCenter does 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 “Understanding Distributed Work Queues” on page 130 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.

Note: For the specifics of configuring work queues, see “Configuring Distributed Work Queues” on page 133 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.

Note: 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 419.

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 <i>last</i> 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 .

Note: For more information on the **Workflows** editor, see “Using the Workflow Editor” on page 419.

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.

Note: It is possible to start a specific workflow with a specific version number. For details, see “Instantiating a Workflow” on page 444.

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, create a new workflow version.
- If you remove an existing step from the workflow, create a new workflow version.
- If you change the step type, for example, from Manual to an automatic step type, 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 Timeout step. ClaimCenter does record the wake-up time (for a Timeout step) that it calculates from the timeout time in the database. However, changing a timeout value does not affect workflows that are already on that step. Therefore, you do not need to create a new workflow version.

If you do modify a workflow, be aware that:

- If you convert a manual step to an automatic step, it can cause issues for an active workflow.
- If you reduce a timeout value, any active workflows that have already hit that step will only wait the previously calculated time.

IMPORTANT If there is an active workflow on a particular step, do **not** alter that step without versioning the workflow.

Workflow Localization

At the start of the workflow execution, the Workflow engine evaluates the workflow locale and uses that locale for notes, documents, templates, and similar items. However, it is possible to set a workflow locale that is different from the default application locale through the workflow editor. This change then affects all notes, documents, templates, email messages, and similar items that the various workflow steps create or use.

You can also:

- Set a different locale for any spawned subworkflows.
- Set a locale for a Gosu block that a workflow executes.
- Set Studio to display a workflow step name in a different locale.

Note: See “Localizing Guidewire Workflow” on page 497 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. Again, see “Localizing Guidewire Workflow” on page 497 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

Note: 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)
```

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 429 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.

- For general information on events, see “Messaging and Events” on page 139 in the *Integration Guide*.
- For what constitutes a valid event name, specifically see “List of Messaging Events in ClaimCenter” on page 156 in the *Integration Guide*.

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 433 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 automatically 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:

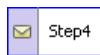
Step ID	Step1
ID	-
To	DefaultOutcome

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 429 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 429 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

Note: 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 <code>SomeUser</code> in <code>SomeGroup</code> . <pre>Workflow.initActivity(Activity) Activity.autoAssign(SomeGroup, SomeUser)</pre> The initialization code creates an activity based on the activity pattern that you set in the Pattern field.

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 439 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 WorkflowTriggerKey.• 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 CancelProcessDate: <code>PolicyPeriod.Cancellation.CancelProcessDate</code>

Note: If the WorkflowTriggerKey typelist does not contain any trigger keys, then you do not see the Trigger option in the dialog.

3. Click on your newly created step. It is possible that there are additional tabs to fill out in the properties area at the bottom of the screen.

Outcome

An **Outcome** is a special step that has no branches leading out of it. It is thus a final or terminal step. If a workflow enters any **Outcome** step, it is complete. It is possible (and likely) for a workflow to have multiple outcomes or final steps.

The **Workflows** editor indicates an outcome step with a gray bar in the box to indicate that this is a final step.



After the Workflow engine successfully enters an **Outcome** step (meaning that the Workflow engine successfully executes the **Enter** block of the **Outcome** step), it does the following:

1. The workflow generates all the listed events and notifications.
2. It executes the **<Finish>** block of the workflow process.
3. It changes the state of the workflow instance to **Completed**.

You *must* structure each workflow script so that its execution eventually and inevitably leads to an **Outcome**. Otherwise, you risk infinitely-running workflows, which means that the load on the Workflow engine can increase linearly over time, crippling performance.

To create a new outcome step

1. Right-click in the workflow workspace, and select **New Outcome**.
2. Enter a step ID in the **New Outcome** dialog.

3. Click on your newly created step. It is possible that there are additional tabs to fill out in the properties area at the bottom of the screen.

Step Branches

A branch is a transition from one step to another. There are multiple kinds of elements that facilitate branching to another step. They are:

- GO
- TRIGGER
- TIMEOUT

The **Workflows** editor indicates a branch by linking two steps with a line and placing one of the following icons on the line to indicate the branch type.

Type	Icon	Description
GO		A branch or transition to another workflow step. It occurs only within an AutoStep workflow step. <ul style="list-style-type: none">• If there is only a single GO branch within the workflow step, branching occurs immediately upon workflow reaching that point.• If there are multiple Go branches within the workflow step, all Go branches (except one) must contain conditional logic. The workflow then determines the appropriate next step based on the defined conditions.
TRIGGER		A branch or transition to another workflow element. It occurs only within a ManualStep workflow step. Branching occurs only upon manual invocation from outside the workflow.
TIMEOUT		A branch or transition to another workflow element. It occurs only within a ManualStep workflow step. Branching to another workflow step occurs only after the passing of a specific time interval.

All branch elements contain a **To** value that indicates the step to which this branch leads. It can also contain an optional embedded Gosu block for the Workflow engine to execute if a workflow instance follows that branch.

How a workflow decides which branch to take depends entirely on the type of the branch. However, the order is always the same:

- The Workflow engine executes the **Enter** block for a given step and generates any events, notifications, and activities (waiting for these activities to complete).
- The Workflow engine attempts to find the first branch that is ready to be taken. It starts with the first branch listed for that step in the outline view, then moves to evaluate the next branch if the previous branch is not ready.
- If no branch is ready (which is possible only on a **ManualStep**), the workflow waits for one to become ready.
- After the Workflow engine selects a branch, it runs the **Exit** block, then executes the Gosu block of the branch.
- Finally, the workflow moves to the next step and begins to evaluate it.

Working with Branch IDs

Every branch also has an ID, which is its reference name. An ID is necessary because the Workflow instance sometimes needs to persist to the database which branch it is trying to execute. (This can happen, for example, if an error occurs in the branch and the workflow drops into the **Error** state). *A branch ID must be unique within a given step.*

If you do not specify an ID for a branch (which occurs frequently), the workflow uses the value of **nextStep** attribute as a default. This works well *except* in the special case in which you have more than one branch leading from the same Step A to the same Step B. (This can happen, for example, if you want to OR multiple conditions together, or if you want different Gosu in the different branches but the same **nextStep**.) In that case, you must

add an ID to each of those branches. Studio complains with a verification error upon loading (or reloading) the workflow scripts if you do not do this.

Do the following to assign an ID to each type of branch:

Type	Action to take
GO	Optionally add an ID to a Go branch. If you do not provide one, Studio defaults the ID to the value of the nextStep 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 WorkflowTriggerKey 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 nextStep 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 GO branch starts.
To	ID of the step on which the GO branch starts.

As discussed (in “Working with Branch IDs” on page 436), 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.

IMPORTANT Unlike all other the IDs used in workflows, `Trigger` IDs are not plain strings but type-list 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 type-list 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 448 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> typelist. 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.

WARNING Do **not** use the current time in a **Time Absolute** expression. The Workflow engine re-evaluates this expression each time it checks **Timeout**. For example, the time-out never ends for the following expression, `java.util.Date.CurrentDate + 1`, as the expression always evaluates to the future.

Creating a Timeout branch. The following graphic illustrate how you define a **Timeout** branch in the **Workflows** editor. To access the **Timeout** branch dialog, right-click the starting step and select **New Timeout** from the menu. Notice that you must enter either time absolute expression or a time delta value. This dialog contains the following fields:

Field	Description
Branch ID	Name you choose for this branch.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.
Time Delta	Time to wait, starting from the time the Workflow instance successfully enters the step.
Time Absolute	Gosu expression that must resolve to a fixed date.

After you create the branch, click on the link that runs between the two steps. You see the following fields:

Field	Description
Branch ID	Automatically generated.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.
Arrow Visible	Show an arrow head on the branch line to indicate direction.
Time Delta	Time to wait, starting from the time the Workflow instance successfully enters the step.
Time Absolute	Gosu expression that must resolve to a fixed date.
Execution	Gosu code to execute if the Workflow engine takes this branch.

Creating New Workflows

To create a new workflow, you can do the following:

Action	Description
Cloning an Existing Workflow	Creates an exact copy of an existing workflow type, with the same name but with an incremented version number. (This process clones the workflow with highest version number, if there multiple versions already exist.) Perform this procedure if you merely want a new version of an existing workflow.
Extending an Existing Workflow	Creates a new (blank) workflow with a name of your choice based on the workflow type of your choice.

Cloning an Existing Workflow

Cloning an existing workflow is a relatively simple process. Also, if you clone an existing, fully built workflow, then you can leverage the work of the original workflow. However, you can only clone existing workflow types. You cannot use this method to create a new workflow type.

To clone an existing workflow

1. Open the **Workflows** node in the **Resources** tree.

2. Select an existing workflow type, right-click and select **New → Workflow Process** from the menu.

Studio creates a cloned, editable copy of the workflow process and inserts 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 **Resources** tree, 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 **Data Model Extensions** node in the **Resources** tree.

6. Right-click the **extensions** folder and select **New → Other file** 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.

8. Open this file, right-click, and choose **Paste** from the menu. Studio pastes in the metadata workflow that you created in a previous step.

9. Add the following line of code to this file (after the **subtype** element):

```
<typekey desc="Language" name="Language" typelist="LanguageType"/>
```

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">
  <typekey desc="Language" name="Language" typelist="LanguageType"/>
</subtype>
```

10. 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**,

11. 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 441. 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 `Resources` tree.
2. Right-click and select `New Workflow Process` from the menu.

Studio opens an outline view and layout view for the new workflow process:

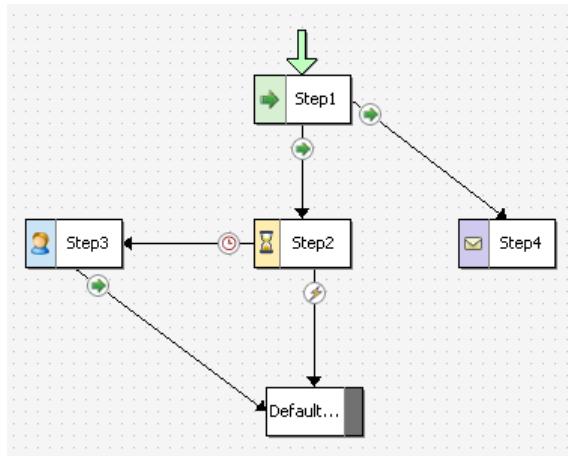
- The outline view contains the few required workflow elements.
- The layout view contains a default outcome (`DefaultOutcome`).

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 427.

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

Note: See “Workflow Debugging and Logging” on page 452 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.

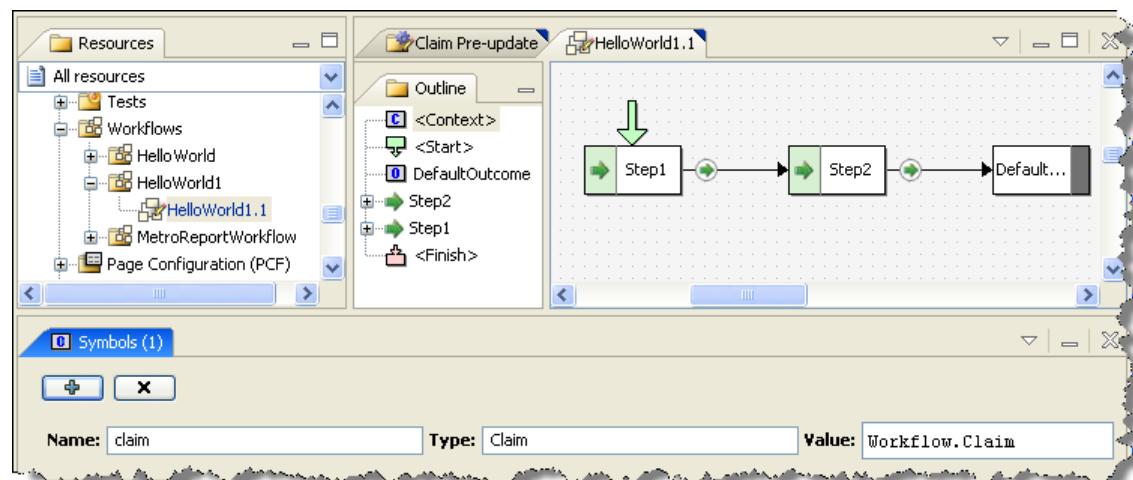
To cause updates to happen, the example instantiates the workflow from an `Edit` screen in ClaimCenter. It then calls a Claim Pre-Update Rule in Studio.

To create a simple workflow and instantiate it

1. Create a `HelloWorld1.eti` file (in `Data Model Extensions → extensions`) 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.

WARNING Never modify files in the `modules/config/...` directories manually. You can cause application damage, preventing the server from starting thereafter. Always use Guidewire Studio to edit and manage ClaimCenter resources stored in XML files.

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 444. 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 450.)

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 “Localize a Workflow” on page 497 for more information.

- b. It steps through each of the workflow steps (meaning that it performs all the actions within that step) until it cannot keep going.
- c. It commits the transaction associated with the executed steps to the database.

Error Handling and Transaction Rollback

If there is an error during a workflow step, the Workflow engine rolls the database back and leaves it in the state that it was. If working with an external system, you need to one of the following:

- You need to design the services in the external system, or,
- You need to use the Guidewire message subsystem to keep an external system state in synchronization with the application database state.

It is important to understand whether a workflow executes synchronously or asynchronously as it affects errors and transaction rollbacks:

Execution type	Application behavior
Synchronous	<p>If any exception occurs during <i>synchronous</i> execution, even after the workflow has gone through several steps, 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).</p> <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	<p>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> <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

Note: See also “Creating a Locale-Specific SubFlow” on page 498.

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.

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.

Note: 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

- “Workflows” on page 410 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 453
- “Pattern Types and Categories” on page 454
- “Using Activity Patterns in Gosu” on page 455
- “Calculating Activity Due Dates” on page 455
- “Defining the Business Calendar” on page 456
- “Configuring Activity Patterns” on page 457
- “Using Activity Patterns with Documents and Emails” on page 459
- “Localizing Activity Patterns” on page 459

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 317.

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, somewhat distinct, operations that you can perform in Gosu involving activity patterns:

- One is 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

To test for a specific activity pattern, use the following Gosu code. This code 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. See also “Found Entities Are Read-only Until Added to a Bundle” on page 148 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" ) )
```

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 keeps 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 255 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 (with no spaces) 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 <code>30 day</code> diary 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
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.

Property	ClaimCenter field	Description
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 457 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 Shared Administration Data” on page 501.

Configuring Localization

Localizing Guidewire ClaimCenter

This topic discusses how to localize your ClaimCenter interface so that it displays information in a language other than the base locale, which is US English.

This topic includes:

- “Understanding Language and Locales” on page 463
- “Working with Localization Configuration Files” on page 464

Understanding Language and Locales

Localization is the act of translating and adapting the viewable text in the ClaimCenter interface so that it is appropriate for use in other countries or languages. Localization allows you to translate the labels for screen names, widgets, and other display elements in the ClaimCenter interface. It also allows you to customize the values shown in the interface so that they are appropriate for your locale. For example, in the United States, the date format is typically set as mm/dd/yyyy. However, in Germany, the date format is typically set as dd/mm/yyyy.

Within ClaimCenter, a locale is the combination of language and formats. You can specify an unlimited number of locales within ClaimCenter. You can also localize the Guidewire development environment, which is Guidewire Studio.

Localizing the Guidewire ClaimCenter application. Within Guidewire ClaimCenter, you can localize the following:

ClaimCenter	See
Activity subjects	“Localizing Shared Administration Data” on page 501

ClaimCenter	See
Date format	"Adding a New Locale" on page 468
Time format	
Number format	
Currency format	
Default application locale	"Setting the Default Application Locale" on page 475
Default user locale	"Setting the User Locale" on page 476
Document, email, note templates	"Localizing Templates" on page 517
Document printing	"Printing in Non-US Character Sets" on page 480
Drop-down lists (typekeys)	"Localizing Typecodes" on page 486
Field labels (display keys)	"Localizing Display Keys" on page 483
Japanese Imperial calendar	"Working with the Japanese Imperial Calendar" on page 541
Workflow	"Setting a Locale for a Workflow" on page 497
Zones	"Configuring Zone Information" on page 476

Localizing the Studio development environment. Within the Guidewire Studio development environment, you can localize the following:

Studio	See
Display keys (field labels)	"Localizing the Development Environment" on page 491
Gosu code	"Setting a Locale for a Gosu Code Block" on page 493
Gosu error messages	"Localizing Gosu Error Messages" on page 494
Rule names and descriptions	"Localizing Rule Set Names and Descriptions" on page 492
Workflow	"Setting a Locale for a Workflow" on page 497
Workflow step names	"Localizing Workflow Step Names" on page 498

Note: To set Guidewire Studio to display double-byte characters, see "Viewing Double-byte Characters in Studio" on page 491.

Working with Localization Configuration Files

To successfully localize Guidewire ClaimCenter, you must work with a number of configuration files. These include:

- XML Configuration Files
- Properties File
- Typelists

XML Configuration Files

The following list describes the main files that you need to configure during the application process.

Configuration files	Use to...	Studio
address-config.xml	Define formats to use for address auto-fill and input masks for postal codes.	Other Resources
currencies.xml	Define the format to use for each separate currency.	Other Resources

Configuration files	Use to...	Studio
config.xml	<p>Set configuration parameters related to localization. These include:</p> <ul style="list-style-type: none"> • DefaultApplicationCurrency—Sets the application currency to use if no other currency is set. • DefaultApplicationLocale—Sets the application locale to use if no other locale is set. See “Setting the Default Application Locale” on page 475. • DefaultCountryCode—Sets The default ISO country code to use if an address does not have a country explicitly set. • PrintFOPUserConfigFile—Set the fully qualified path to a valid FOP user configuration file. See “Printing in Non-US Character Sets” on page 480. • PrintFontFamilyName—Sets the name of the font family for the custom fonts as defined in the FOP user configuration file. See “Printing in Non-US Character Sets” on page 480. 	Other Resources
fieldvalidators.xml	Format field validators for phone numbers, ID fields, money fields, and other fields that need validation and input masks.	Data Model Extensions
localization.xml	Add a new locale entry (GWLocale) with appropriate settings for number, date, time, and currency formats. Any language code that you add must match that of a language code defined in the LanguageType typelist.	Other Resources
zone-config.xml	Define zone information to use for region and address auto-fill	Other Resources

Properties File

The following list describes the properties files that you need to create during the application localization process.

Properties files	Use to
display.properties	Create localized versions of the display keys and type codes to use within Guidewire ClaimCenter.
	IMPORTANT At the very least, a locale must have a folder specific to that locale in the <i>ClaimCenter/modules/configuration/config/locale</i> folder. This folder must contain a <i>display.properties</i> file. The file can be empty, but it must exist. Otherwise, ClaimCenter does not recognize the locale.
gosu.display.properties	Create localized versions of Gosu error messages.
studio.display.properties	Create localized versions of Studio field labels. You can translate the labels into the language of your choice.

Typelists

The following list describes the main typelists that you need to configure during the localization process.

Typelist	Use to
Currency	Define new currency codes
Jurisdiction	Define new jurisdictions
LanguageType	Define new language codes
State	Define new state codes
ZoneType	Define new zone regions

Working with Locales

This topic discusses how to add additional locales to your base application.

This topic includes:

- “Locale Configuration Files” on page 467
- “Adding a New Locale” on page 468

Locale Configuration Files

A locale is the combination of language and formats (for example, time, date, and currency formats). You can specify an unlimited number of locales within ClaimCenter. The following list describes the key ClaimCenter resources for configuring locales:

File	Purpose
config.xml	Contains configuration parameters related to localization. These include: <ul style="list-style-type: none">• DefaultApplicationLocale—Sets the application locale to use if no other locale is set.• PrintFOPUserConfigFile—Set the fully qualified path to a valid FOP user configuration file.• PrintFontFamilyName—Sets the name of the font family for the custom fonts as defined in your FOP user configuration file. See “Printing in Non-US Character Sets” on page 480.
display.properties	Contains a listing of the display keys and typecodes to use within Guidewire ClaimCenter. Each locale must have a separate <code>display.properties</code> file. It is the presence of multiple <code>display.properties</code> files that alerts ClaimCenter to the fact that multiple locales exist. <p>The <code>display.properties</code> file (as with all property files) is a standard Java properties file with the following format:</p> <pre>display_key_name = value</pre>

File	Purpose
fieldvalidators.xml	Contains format information for fields such as currencies, phone numbers, and ID fields, and other fields that need validation and input masks. If you create multiple locales, then you must update this file to accommodate all possible formats for all possible locales. To use the Money field validator as an example, you must update the regular expression for this field if you add a comma (,) as a possible decimal separator.
	IMPORTANT You cannot use a single regular expression on a field that varies by locale, unless you use the <code>LocalizedString</code> data type for that field. See “Localizing Field Validators” on page 507 for details.
gosu.display.properties	Contains Gosu error messages. Studio displays these messages if it encounters a Gosu error condition. You can translate these error messages into the language of your choice.
LanguageType typelist	Typelist that defines the locale choices available in the ClaimCenter interface and within Studio. See “Adding a New Locale” on page 468.
localization.xml	Defines the locales available in ClaimCenter. See “Adding a New Locale” on page 468.
studio.display.properties	Contains Studio field labels. You can translate the labels into the language of your choice. To view a different locale in Studio, select it from the drop-down list of defined locales in the bottom right-hand corner of the Studio screen.
zone-config.xml	Use to specify the zones for each country. Zones are address components used for address autofill and region creation.

Adding a New Locale

A locale defines ClaimCenter behavior for a particular country and language. Creating a locale enables its use and defines its date, time, number, and currency formats. To add a locale to the base ClaimCenter configuration, perform the following steps:

- Step 1: Add the Locale to the Localization File
- Step 2: Add the Locale to the Language Type Typelist
- Step 3: Add the Locale to the Collations File
- Step 4: Create and Populate the New Locale Folder

Step 1: Add the Locale to the Localization File

You define and manage locales in Guidewire Studio. To edit the locales, expand **Other Resources**, and then click `localization.xml`. This file initially contains the base ClaimCenter locale, which is US English. You can modify or remove this locale, or add additional new locales to the file.

As an example, the following is the definition of the base US English locale:

```
<GWLocale name="English (US)" code="en_US" typecode="en_US">
    <DateFormat short="MM/dd/yyyy"
        medium="MMM d, yyyy"
        long="E, MMM d, yyyy" />
    <TimeFormat short="hh:mm aa"
        medium="hh:mm aa"
        long="hh:mm aa"/>
    <NumberFormat decimalSymbol="."
        thousandsSymbol="," />
    <CurrencyFormat positivePattern="$#"
        negativePattern="($#)"
        zeroValue="-" />
</GWLocale>
```

To create a French locale, for example, add another <GWLocale> element, setting the code similar to the following, with the appropriate date, time, number, and currency formats.

```
<GWLocale name="French (FR)" code="fr_FR" typecode="fr_FR">
  ...
</GWLocale>
```

The <GWLocale> Element

The <GWLocale> element defines a locale. This element has multiple parameters—name, code, and typecode:

- The name parameter is a description of the locale. You use it only for reference.
- The code parameter is the locale identifier and must follow the Java standard for locale names. This standard states that you can specify names in an xx, xx_yy, or xx_yy_zzz format. The components of these formats are:

xx	Required	The two-character lowercase ISO 639-1 code for that language. To view a list of valid codes, refer to following web site: http://www.loc.gov/standards/iso639-2/php/English_list.php
yy	Optional	A two-character uppercase ISO 3166-1-alpha-2 code for a country. To view a list of countries, refer to the following web site: http://www.iso.org/iso/english_country_names_and_code_elements
zzz	Optional	A custom string. For example, you can use this to define regions within a country. You can make the string any length.

- The typecode parameter is the ClaimCenter typecode for the locale. Use this parameter to access the locale from within ClaimCenter, for example, to assist in assigning different users to different locales. If you add a new locale, then you must create a new entry in the LanguageType typelist.

Specifying format elements. Within a <GWLocale> element, you can specify local formats for the following:

- date (<DateFormat>)
- time (<TimeFormat>)
- number (<NumberFormat>)
- currency (<CurrencyFormat>)

<DateFormat>

<TimeFormat>

The <DateFormat> and <TimeFormat> subelements on <GWLocale> specify how Guidewire ClaimCenter formats and displays dates and times. You can specify any of the following attribute values for the <DateFormat> and <TimeFormat> elements:

- short
- medium
- long

For example (for the en_us locale):

```
<DateFormat short="MM/dd/yyyy"
            medium="MMM d, yyyy"
            long="E, MMM d, yyyy" />

<TimeFormat short="hh:mm aa"
            medium="hh:mm aa"
            long="hh:mm aa"/>
```

In general, you can use any of the date and time patterns supported by the Java class `SimpleDateFormat` for the `medium` and `long` formats. However, Guidewire maps the `short` format to the date picker widget, which does not support arbitrary date formats. For a description of these patterns, refer to the Java documentation at the following web site:

<http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html>

ClaimCenter generally uses the short form to recognize dates entered by the user. It generally uses the other forms to display dates and times.

Using the 'short' Format

Only define patterns for the short date and time definitions that result in a fixed-length output that matches the pattern length. In general, dates have three components, year, month, and day, each defined by a specific pattern. Each pattern provides a fixed-width output. For example, the following patterns all provide a fixed-width output. In this case, each output contains the same number of characters as the format pattern.

Format	Pattern	Output
year	yyyy	4 digit output, fixed
month	MM	2 digit output, fixed
day	dd	2 digit output, fixed

The following list describes an incorrect (non-fixed-width) pattern:

Format	Pattern	Output
year	yyyy	4 digit output, fixed
month	MMM	variable length output
day	dd	2 digit output, fixed

The pattern MMM does not work as there are languages in which the abbreviated month string is not three characters in length. Attempting to use this pattern with a language in which there are abbreviated month strings that are not three characters in length can cause a validation error.

<NumberFormat>

The <NumberFormat> subelement on <GWLocale> determines how Guidewire ClaimCenter displays numbers:

- The thousandsSymbol attribute separates groups of three digits to the left of the decimal mark. This is typically a comma, period, or space.
- The decimalSymbol attribute separates a whole number from its decimal fraction. This is typically a period or comma.

For example, the base ClaimCenter configuration defines the <NumberFormat> element such that the thousands separator is a period and the decimal separator is a comma:

```
<NumberFormat decimalSymbol="." thousandsSymbol="," />
```

If you change the decimalSymbol attribute on <NumberFormat>, then you must also modify the validator definition for Money in `fieldvalidators.xml`. In the base configuration, the validator definition for Money looks similar to the following:

```
<ValidatorDef description="Validator.Money"
    input-mask=""
    name="Money"
    value="-?[0-9]{0,16}(\.[0-9]{0,2})?"/>
```

If you specify a new decimal separator, then remove the regular expression of the Money field validator in `fieldvalidators.xml`. Replace it, instead, with a field validator that includes range values for ceiling and floor, for example:

```
<ValidatorDef description="Validator.Money"
    input-mask=""
    name="Money"
    ceiling="9999999999999999.99"
```

```
    floor="-999999999999999.99"
    value=".*/>
```

IMPORTANT In `fieldvalidators.xml`, use regular expressions with `String` values only. Use `floor` and `ceiling` range values for numeric fields.

See “`<ValidatorDef>`” on page 295 for more information on field validators.

`<CurrencyFormat>`

IMPORTANT It is important that you **always** display currency values correctly for the current locale. For example, if using a single currency, \$10 in English can be represented as 10 USD in German. In both of these cases, however, the currency unit is dollars. It would be a serious misrepresentation of your financial data to use a \$ symbol for one locale and a € symbol for another. (10 dollars is not equal to 10 euros, at least not at the current exchange rate.)

You use the `<CurrencyFormat>` element to control how ClaimCenter displays currencies.

<code>positivePattern</code>	The <code>positivePattern</code> and <code>negativePattern</code> parameters define how ClaimCenter displays positive and negative currency values. ClaimCenter renders the character # (pound sign) with the actual number, but display all other characters in the pattern literally. For example, the <code>positivePattern</code> <code>#\$</code> displays the value 32 as \$32, and the <code>negativePattern</code> <code>(\$#)</code> displays the value -5 as (\$5). The pattern can include more than one character, such as <code># USD</code> to display as 5 USD.
<code>zeroValue</code>	The <code>zeroValue</code> parameter defines how ClaimCenter displays zero. For example, this can be 0 (zero) or — (dash). (ClaimCenter displays money or currencyamount fields that contain null as empty or blank.)

Currency modes. Guidewire ClaimCenter provides the ability to use a single currency throughout the application or to use multiple currencies.

<code>SINGLE mode</code>	To set the application to a single currency, you must set configuration parameter <code>DefaultApplicationCurrency</code> to that currency and set <code>MultiCurrencyDisplayMode</code> to <code>SINGLE</code> . In this mode, ClaimCenter uses the <code>CurrencyFormat</code> entries in each <code>GWLocale</code> in <code>localization.xml</code> to format the money amount, depending on the each user's current Locale. As all money values are in the default currency, you must ensure that the <code>CurrencyFormat</code> for each <code>GWLocale</code> specifies the money format for that one currency. It is possible to set slightly different formatting based on local custom, but all money formatting must be for the one default currency.
<code>MULTIPLE mode</code>	To set the application to multiple currencies, you must set configuration parameter <code>MultiCurrencyDisplayMode</code> to <code>MULTIPLE</code> . In <code>MULTIPLE</code> mode, ClaimCenter obtains money formatting information from <code>currencies.xml</code> . In this mode, ClaimCenter ignores any <code>CurrencyFormat</code> information in <code>localization.xml</code> . However, even though unused, a tag for the default currency must be present in file <code>localization.xml</code> , even in <code>MULTIPLE</code> mode.

Currency overrides. It is possible to override the default currency type and format by using the following tags:

- `<CurrencyTypeOverride>`
- `<CurrencyFormatOverride>`

You use these tags in file `localization.xml` to override the default currency type and the `<CurrencyFormat>` tag for that currency set in `currencies.xml`. The following example illustrates this concept:

```
<GWLocale>
  ...
  <!-- Still required in MULTIPLE mode -->
  <CurrencyFormat ../>
  <CurrencyTypeOverride code="eur">
    <CurrencyFormatOverride positivePattern="# euro" negativePattern="-# euro"/>
  </CurrencyTypeOverride>
  ...
</GWLocale>
```

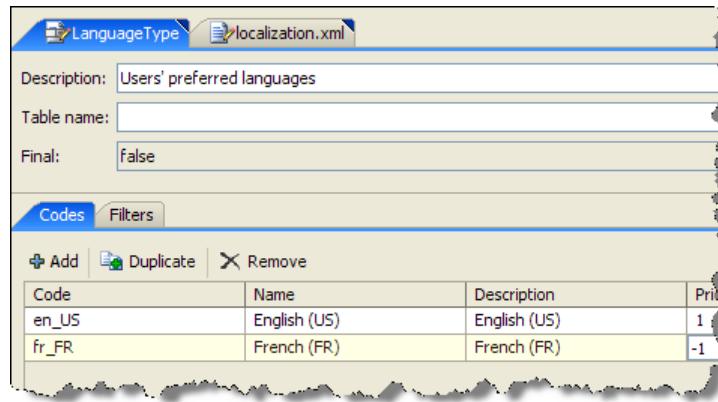
Step 2: Add the Locale to the Language Type Typelist

After you add a new locale to `localization.xml`, you must then add a typecode for that locale to the `LanguageType` typelist. For example, if you create a new `fr_FR` locale, then you need to add this locale to the `LanguageType` typelist. (For information on working with typelists, see “Working with Typelists” on page 315.)

To add a locale, click **Add** and fill in the appropriate fields.

Field	Description
Code	Associates the entry with a locale. The code must match the <code>code</code> attribute of an existing <code>GWLocale</code> definition.
Name	The name of the locale as you want it to appear in the ClaimCenter interface.
Description	A longer description of this typecode. The maximum description size is 512 characters.
IMPORTANT You must enter a typecode description if you add a new typecode. If you do not:	
• Studio generates error messages but continues to start.	
• The application server generates error messages and refuses to start.	

For example, if you add a locale for French (`fr_FR`), you see the following:



Step 3: Add the Locale to the Collations File

Guidewire requires that you set the *collation strength* for every locale that you define. Collation strength refers to how selective (or restrictive) the search and sort code treats the collation process. For example, collation strength controls whether the search and sort code respects or ignores differences in case and accent on a character (such as the leading character on a word).

You set the database collation name corresponding to each collation strength in file `collations.xml`. This file must contain a collation entry for every defined locale. Depending on the locales that you have defined, this file looks similar to the following.

```
<?xml version="1.0" encoding="UTF-8" ?>
<Collations xmlns="http://guidewire.com/collations">
  <Database type="ORACLE">
    <Collation locale="en" primary="BINARY_AI" secondary="BINARY_CI" tertiary="BINARY" />
    <Collation locale="fr" primary="FRENCH_M_AI" secondary="FRENCH_M_CI" tertiary="FRENCH_M" />
    <Collation locale="ja" primary="JAPANESE_M_AI" secondary="JAPANESE_M_CI" tertiary="JAPANESE_M" />
    <Collation locale="de" primary="GERMAN_AI" secondary="GERMAN_CI" tertiary="GERMAN" />
    <Collation locale="ru" primary="GENERIC_M_AI" secondary="GENERIC_M_CI" tertiary="GENERIC_M" />
  </Database>
  ...
</Collations>
```

Note: There is an equivalent series of entries for SQL Server, if you are using that database.

If an entry does not exist for your defined locale, then you must add it to `collations.xml` under the proper `<Database>` element. Use the existing entries as a model of what to enter. See “Localized Search and Sort” on page 525 for details on collation strength and how the different database types treat this value differently.

If you do not define the locale in `collations.xml`, then ClaimCenter does the following:

- In development mode, ClaimCenter logs a warning message at application server start up.
- In production mode, ClaimCenter logs an error message and the application server refuses to start.

IMPORTANT Guidewire defines locales in `collations.xml` using the two letter language code only. This is sufficient unless you wish to differentiate between countries that speak the same language. For example, if you add a locale for Great Britain (`en_GB`), then the `en` locale defined in the base configuration `collations.xml` file is sufficient to cover both U.S. English and British English. However, if you want to define different search or sort algorithms for Great Britain, then you need to add the `en_GB` locale to `collations.xml` and modify its attribute definitions accordingly.

Example Collations Settings for Spanish

Guidewire suggests that following collation settings for the Spanish `es` locale. Using a locale of `es` handles any locale that uses the Spanish language.

For Oracle. Add the following to `<Database type="ORACLE">`:

```
<Collation locale="es" primary="SPANISH_AI" secondary="SPANISH_CI" tertiary="SPANISH"/>
```

For SQL Server. Add the following to `<Database type="SQL_SERVER">`:

```
<Collation locale="es" primary="Modern_Spanish_CI_AI" secondary="Modern_Spanish_CI_AS"
tertiary="Modern_Spanish_CS_AS"/>
```

Alternately, if you want to handle the following special cases, use `XSPANISH` instead of `SPANISH`:

- `ch`
- `Ch`
- `CH`
- `ll`
- `Ll`
- `LL`

Step 4: Create and Populate the New Locale Folder

For each locale that you create, you also need to create a folder for that locale in the following directory:

```
modules/configuration/config/locale
```

For example, if you add a French locale for France (`fr_FR`), then you need to create the following folder:

```
modules/configuration/config/locale/fr_FR
```

After you create this folder, you must populate it with the following files, properly translated for that locale. Shut down Studio before you start as you need to modify your copy of `display.properties` before starting Studio again. Otherwise, Studio generates an error.

You can obtain copies of the necessary files at the listed location. Each locale folder must contain a `display.properties` file, at the very least, even if it is empty. It is the presence of multiple copies of the `display.properties` file that alerts ClaimCenter to the fact that multiple locales exist.

The following table lists the various display properties files and their location. In some cases, there are multiple copies of a file, one for system data and one for application-specific data. *If there are multiple copies of a file, then you need to merge the two copies to obtain a complete list of display keys and typecodes for that file type.*

IMPORTANT Only modify files in `modules/configuration`. Otherwise, you can damage your ClaimCenter installation, causing it not to start thereafter.

File	Copy from	Description
<code>display.properties</code>	<code>modules/cc/config/locale/en_US</code> <code>modules/pl/config/locale/en_US</code>	Contains ClaimCenter interface display keys and typecodes.
<code>gosu.display.properties</code>	<code>modules/pl/config/locale/en_US</code>	Contains Gosu error messages.
<code>studio.display.properties</code>	<code>modules/cc/config/locale/en_US</code> <code>modules/pl/config/locale/en_US</code>	Contains Studio labels.

Lastly, open your copy of `display.properties`. This file contains the following variable:

`{Global.AppName}`

If a definition for this variable does not exist in the file, then you must define it. Otherwise, Studio generates an error upon start up. For example:

`Global.AppName = ClaimCenter`

Thereafter, ClaimCenter replaces `{Global.AppName}` in the following expression with the correct value.

`JSP.Login.Login.Title = Guidewire {Global.AppName}`

or

`JSP.Login.Login.Title = Guidewire ClaimCenter`

You must restart Studio for your changes to take effect.

IMPORTANT If you are upgrading your application, be sure to review the related ClaimCenter release notes. It is possible for Guidewire to add additional display keys as part of the application upgrade. If so, you must add the new display keys to the `display.properties` files (including those for Studio and Gosu) before you localize these files.

Localizing the ClaimCenter Interface

This topic discusses ways to localize the Guidewire ClaimCenter application interface.

This topic includes:

- “Setting the Default Application Locale” on page 475
- “Setting the User Locale” on page 476
- “Configuring Zone Information” on page 476
- “Setting the IME Mode for Field Inputs” on page 480
- “Printing in Non-US Character Sets” on page 480

Setting the Default Application Locale

If ClaimCenter cannot determine which locale is appropriate for a user, it uses the default application locale. To specify the default locale, set the `DefaultApplicationLocale` configuration parameter (in `config.xml`). The parameter value must match the `code` attribute of one of the `GWLocale` definitions. The following example sets the default locale to German:

```
<param name="DefaultApplicationLocale" value="de_DE" />
```

If ClaimCenter cannot find a `GWLocale` with the matching code, it uses `en_US` as the default locale.

Working with a default locale in Gosu. Within Gosu, you can determine the default locale by invoking the following method:

```
gw.api.util.LocaleUtil.getDefaultLocale().Code
```

Setting the User Locale

As a user logs in or navigates through the interface, ClaimCenter determines the appropriate local based on the following:

- As ClaimCenter authenticates the current user during the log-in process, it chooses the individual locale set for that user, if one exists.
- If ClaimCenter cannot locate an individual locale setting for that user, it determines the locale by the Web browser. ClaimCenter chooses the `GWLocale` definition whose `code` attribute matches the value of the `Accept-Language` HTTP header for that browser. If the `Accept-Language` header contains multiple codes, then ClaimCenter uses the first matching code.
- If ClaimCenter cannot determine the locale, it uses the default locale, which is `en_US`.

To set a default user locale

1. On the **ClaimCenter Administration** tab, search for the user whose profile you want to modify.
2. On the **Profile** page for that user, click **Edit**.
3. Set the desired language using the language selector drop-down list.

To set your own locale

If multiple locales exist, then a language selector appears along the top of the ClaimCenter interface. To set the locale, select a value from the drop-down list.

To add a check mark to the language selector

It is possible to add a check mark to the active selection in the language selector. ClaimCenter uses the `checked` attribute in the `MenuItem` widget—as part of the `LanguageTabBarLink` widget, which, in turn, is part of the `TabBar` widget—to create the check mark.

To display a check mark next to the active language selection, set the `checked` attribute to the following:

```
util.LocaleUtil.getCurrentUserLanguage() == lang
```

Configuring Zone Information

Guidewire ClaimCenter uses file `zone-config.xml` to define one or more zones. A *zone* is a combination of a country and zero-to-many address elements such as a city, or state (United States), or province (Canada), for example. You can configure zones to apply to any area within a single country. In the United States, for example, you typically define zone type by state, city, county, and ZIP (postal) code.

ClaimCenter uses zones for the following:

- Region and address auto-fill
- Business week and holiday definition by zone

You access `zone-config.xml` through Studio in the **Other Resources** folder. Within this file, you define the links between zones (the zone hierarchy) and how ClaimCenter is to use the links to extract the value of zone from address data.

In the base configuration, ClaimCenter defines the zone hierarchy for the United States using the following elements:

```
<Zones countryCode="US">
  <Zone code="zip" fileColumn="1" regionMatchOrder="1" granularity="1" unique="true">
    <AddressZoneValue>Address.PostalCode.substring(0, 5)</AddressZoneValue>
    <Links>
      <Link toZone="city"/>
    </Links>
  </Zone>
```

```

<Zone code="state" fileColumn="2" orgZone="true" regionMatchOrder="3" granularity="4">
  <AddressZoneValue>Address.State.Code</AddressZoneValue>
  <Links>
    <Link toZone="zip" lookupOrder="1"/>
    <Link toZone="county"/>
    <Link toZone="city"/>
  </Links>
</Zone>
<Zone code="city" fileColumn="3" granularity="2">
  <ZoneCode>state + " + city</ZoneCode>
  <AddressZoneValue>Address.State.Code + " + Address.City</AddressZoneValue>
</Zone>
<Zone code="county" fileColumn="4" regionMatchOrder="2" granularity="3">
  <ZoneCode>state + " + county</ZoneCode>
  <AddressZoneValue>Address.State.Code + " + Address.County</AddressZoneValue>
  <Links>
    <Link toZone="zip" lookupOrder="1"/>
    <Link toZone="city" lookupOrder="2"/>
  </Links>
</Zone>
</Zones>

```

File zone-config.xml

File `zone-config.xml` contains the following XML elements:

- `<Zones>`
- `<Zone>`
- `<ZoneCode>`
- `<Links>`
- `<AddressZoneValue>`

See Also

- “[zone_import Command](#)” on page 178 in the *System Administration Guide*.

`<Zones>`

This element defines the largest region, a country. The `zone-config.xml` files contains one or more country definitions, one for each `<Zones>` element. Each `<Zones>` element contains zero or more `<Zone>` elements. Even though Guidewire does not strictly require that you provide one or more `<Zone>` elements, you most likely want at least one zone element. For example, it is possible—but not recommended—to create the following:

```
<Zones countryCode="AU"/>
```

However, in almost all cases, you want to add one or more `<Zone>` subelements to the `<Zones>` element.

The `<Zones>` element takes the following attributes:

Attribute	Description
<code>countryCode</code>	<i>Required.</i> Defines the country to which this zone configuration applies.
<code>dataFile</code>	For Guidewire internal use only. Ignore.

`<Zone>`

This element defines a zone type. The zone type must exist in the `ZoneType` typelist. The `Zone` element takes the following attributes, all optional except for the `code` attribute:

Attribute	Description
<code>code</code>	Sets the zone type. This must be a valid value defined in the <code>ZoneType</code> typelist. You also use this value as a symbol in <code><ZoneCode></code> expressions to represent the data extracted from the data import file based on the column specified in the <code>fileColumn</code> attribute.

Attribute	Description
fileColumn	<p><i>Optional.</i> Specifies the column in the import data file from which to extract the zone data. The numeric value of <code>fileColumn</code> indicates the ordered number of the column in the data file.</p> <p>Frequently (but not exclusively), a <code><Zone></code> element without a <code>fileColumn</code> attribute contains a manipulation of data imported from one or more other <code><Zone></code> definitions. For example, in the base configuration, Guidewire defines the following <code>fsa</code> zone in the Canadian <code><Zones></code> element:</p> <pre><Zone code="fsa" regionMatchOrder="1" granularity="1"> <ZoneCode> postalcode.substring(0,3) </ZoneCode> <AddressZoneValue> Address.PostalCode.substring(0,3) </AddressZoneValue> </Zone></pre> <p>Notice that both the <code><ZoneCode></code> and <code><AddressZoneValue></code> elements extract data from the actual address data by parsing the data into substrings.</p> <p>Note:</p> <ul style="list-style-type: none"> You need to specify at least one <code><Zone></code> element with a <code>fileColumn</code> attribute to trigger the import of data from the address zone data file. If you do not specify at least one <code>fileColumn</code> attribute, then ClaimCenter does not import data from the address zone data file. You need to import address zone data upon first installing Guidewire ClaimCenter and then at infrequent intervals thereafter as you receive data updates.
granularity	Sets the level of granularity for each defined zone. The smallest geographical region is 1. The next larger geographical region is 2, and so on. The sequence of numbers must be continuous. ClaimCenter uses this value with holidays and business weeks. For ClaimCenter Catastrophe administration, you must set the granularity for zones that you want to make available in the list of Zone Types on the New Catastrophe page.
orgZone	For Guidewire internal use only. Ignore.
regionMatchOrder	<p>Controls the order in which ClaimCenter uses these zones in matching algorithms. ClaimCenter uses this attribute as it matches users to a zone for location- or proximity-based assignment. For example, in the base configuration, Guidewire defines the following <code><Zone></code> for county:</p> <pre><Zone code="county" fileColumn="4" regionMatchOrder="2" granularity="3"> ... </Zone></pre> <p>Setting the <code>regionMatchOrder</code> to 2 means that ClaimCenter matches county data second, after another zone, while matching a user to a location.</p> <p>Notice also that the county value:</p> <ul style="list-style-type: none"> Appears in the fourth column of the data file. Is third in granularity, one size less than a state (<code>granularity</code> 4) and one size more than a city (<code>granularity</code> 2).
unique	<i>Optional.</i> Specifies whether this zone data is unique. For example, in the United States, a county data value by itself does not guarantee uniqueness across all states. (There is a county with the name of <i>Union</i> in seventeen states and a county with the name of <i>Adams</i> in twelve states.) In situations such as this, use a <code><ZoneCode></code> element to construct a zone expression that uniquely identifies that zone data.

<ZoneCode>

It is possible for zone information to *not* be unique, meaning that the zone import data column contains two or more identical values. If this is the case, then you need to build an expression to uniquely identify the individual zone data using symbols.

For example, in the United States, it is possible for multiple states to have a city with the same name (Portland, OR and Portland, ME). Thus, to uniquely identify the city, you need to associate a particular state with the city as well. To do this, you create an expression that prepends the state import data value to the city import data value to obtain a unique city-state code. For example:

```
<ZoneCode>state + ":" + city</ZoneCode>
```

Only use values to construct the expression that are valid <Zone> code values (other than constants) that you defined within a <Zones> element for this country.

<Links>

This element defines one or more <Link> elements, each of which defines a connection (a link) between one zone type and another. For example, in the base configuration, Guidewire provides a <Link> element that defines a link between a county and a ZIP code in the United States. You can also use a link to define a lookup order to speed up searches.

The <Link> element contains the following attributes:

Attributes	Description
lookupOrder	<i>Optional.</i> Tells the application in what order to apply this value while looking up users by zone. This can increase performance somewhat. If you do not specify a <code>lookupOrder</code> value, ClaimCenter uses the order as it appears in the file.
toZone	<i>Required.</i> Defines a relationship to another zone (region).

<AddressZoneValue>

This optional element uses an expression to define how to extract the zone code from an address. Use entity dot notation, such as `Address.PostalCode`, to define a value for this element. These expressions can be subsets of an element or a concatenation of elements.

ClaimCenter extracts a value from the address data that uses this element and matches the value against zone data in the database. For example, in the base configuration, ClaimCenter defines a <Zone> element of type `postalcode` for Canada. It looks similar to the following:

```
<Zones countryCode="CA">
  <Zone code="postalcode" fileColumn="1" unique="true">
    <AddressZoneValue> Address.PostalCode </AddressZoneValue>
  </Zone>
```

This definition indicates that ClaimCenter is to use the value of the `PostalCode` field on the `Address` entity as the value of the `postalcode` zone.

Note: Guidewire recommends that you set this value even if there is a property defined on the `Address` entity that has the same name as the `Zone` name.

Base Zone Hierarchies

In the base configuration, Guidewire defines zone hierarchies for the United States and Canada. You can add additional country zone definitions to `zone-config.xml`. Only one country zone at a time is active in ClaimCenter.

In the base configuration, ClaimCenter uses zone information with the following data objects:

- `BusinessWeek`
- `Catastrophe`
- `Holiday`
- `Region`

See Also

- “Understanding Autofill and Zone Mapping” on page 61 in the *Contact Management Guide*

Setting the IME Mode for Field Inputs

Guidewire ClaimCenter provides three states for `imeMode`, which you set at the *field* level in Guidewire Studio. The three possible states are:

State	Description
Active	<ul style="list-style-type: none">Japanese – Turns on the last entry type that you selected (Hiragana, or full-width Katakana, for example)Chinese – Turns on pinyin entry
Inactive	<ul style="list-style-type: none">Japanese – Turns off Roman entryChinese – Turns on Roman entry
Not selected	<ul style="list-style-type: none">Does nothing – Leaves the currently-selected IME mode as set

You set the `imeMode` at the field level on a PCF. For a *rōmaji* field in Japanese, for example, you might set `imeMode` to `inactive`. Then, on the next field (that needs Kanji entry, for example), you would set `imeMode` to `active`. And so on for the various fields on the PCF. In general, however, you only need set `imeMode` on text input fields and possibly drop-down fields such as typelists and range input fields.

It is important to understand that ClaimCenter does not actually set the actual IME mode. ClaimCenter merely turns IME on or off for each field. In other words, ClaimCenter cannot dictate that a certain field must contain Zenkaku Katakana and a different field must contain Hiragana. The choice of input conversion style is left to the user.

Printing in Non-US Character Sets

The Guidewire printing infrastructure fully supports internationalization. To use non-supported character sets (such as Cyrillic, for example), you must perform some configuration of Apache FOP (Formatting Objects Processor), however. This section provides general directions for performing such configuration, and provides example configuration file excerpts in support of the Cyrillic character set.

Overview of Required FOP Changes

This example uses the following configuration (`config.xml`) parameters:

Configuration parameter	Description
<code>PrintFOPUserConfigFile</code>	The fully qualified path to a valid FOP user configuration file.
<code>PrintFontFamilyName</code>	The name of the font family for the custom fonts as defined in your FOP user configuration file.

Suppose that you want to print PDFs in Russian, which uses the Cyrillic character set. The default font for PDF generation does not support this character set. Therefore, you need to customize FOP so that it uses a set of fonts that does support the Cyrillic character set. Fortunately, the generic Windows Arial TrueType font family (normal, bold, italic, bold-italic) does support Cyrillic.

This simple example assumes the following:

- The Apache FOP application 0.95 exists on your machine.
- The `fop.jar` is on the class path.
- The Arial fonts exist in `C:\WINDOWS\Fonts`.
- The fonts are TrueType fonts.

Note: The process for non-TTF FOP supported fonts is slightly different. Refer to the Apache FOP documentation for more information.

Configuring Apache FOP for Cyrillic

To support print functionality in a non-U.S. encoding, perform the following steps:

- Step 1: Configure TTFRReader
- Step 2: Generate FOP Font Metrics Files
- Step 3: Register Your FOP Configuration and Font Family
- Step 4: Register the Fonts in the FOP Configuration File
- Step 5: Test Your Configuration

Step 1: Configure TTFRReader

You must use Apache FOP 0.95. You can download this version of Apache FOP from the following web site:

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

After you download and install Apache FOP, do the following:

1. Make a copy of `fop.bat` (in the root installation directory) and rename it `ttfreader.bat`.
2. Open `ttfreader.bat` and change the last line to read:

```
"%JAVACMD%" "%JAVAOPTS% %LOGCHOICE% %LOGLEVEL% -cp "%LOCALCLASSPATH%"  
org.apache.fop.fonts.apps.TTFRReader %FOP_CMD_LINE_ARGS%
```

Essentially, you are changing `org.apache.fop.cli.Main` to `org.apache.fop.fonts.apps.TTFRReader`. You need to do this step so that the code to generate the font metrics works correctly.

Step 2: Generate FOP Font Metrics Files

FOP requires font metrics to use a font. You must give it the font metrics. To generate the font metrics, run the following commands. These commands activate the Apache FOP TTFRReader.

```
ttfreader.bat -enc ansi C:\WINDOWS\Fonts\arial.ttf C:\fopconfig\arial.xml  
ttfreader.bat -enc ansi C:\WINDOWS\Fonts\ariali.ttf C:\fopconfig\ariali.xml  
ttfreader.bat -enc ansi C:\WINDOWS\Fonts\arialbi.ttf C:\fopconfig\arialbi.xml  
ttfreader.bat -enc ansi C:\WINDOWS\Fonts\arialbd.ttf C:\fopconfig\arialbd.xml
```

This generates metrics files for the Arial font family and stores those metrics in the `C:\fopconfig` directory. Do not proceed until you see the following files in the `fopconfig` directory:

```
arial.xml  
arialbd.xml  
arialbi.xml  
ariali.xml
```

Step 3: Register Your FOP Configuration and Font Family

Open your application configuration file:

`ClaimCenter/modules/configuration/config/config.xml`

Find the following parameters and set them accordingly:

Parameter	Value
<code>PrintFontFamilyName</code>	Cyrillic
<code>PrintFOPUserConfigFile</code>	<code>C:\fopconfig\fop.xconf</code>

WARNING Only modify `config.xml` in the `ClaimCenter/modules/configuration` directory. Otherwise, you can invalidate your Guidewire ClaimCenter installation.

Step 4: Register the Fonts in the FOP Configuration File

To create a user configuration file for FOP, perform the following steps:

1. Copy the following file:

C:\fop-0.95\conf\fop.xconf

into the following directory:

C:\fopconfig\

IMPORTANT The name of the FOP configuration file must be the same as that set for the PrintFOPUserConfigFile parameter described in Step 3: Register Your FOP Configuration and Font Family.

2. Open fop.xconf in an XML editor and find the <fonts> section. It looks similar to the following:

```
<fonts>
  ...
</fonts>
```

3. Enter the following font information in the appropriate place.

```
<fonts>
  <font metrics-url="c:\\fopconfig\\arial.xml" kerning="yes" embed-file="arial.ttf">
    <font-triplet name="Cyrillic" style="normal" weight="normal"/>
  </font>
  <font metrics-file="c:\\fopconfig\\arialbd.xml" kerning="yes" embed-file="arialbd.ttf">
    <font-triplet name="Cyrillic" style="normal" weight="bold"/>
  </font>
  <font metrics-url="c:\\fopconfig\\ariali.xml" kerning="yes" embed-file="ariali.ttf">
    <font-triplet name="Cyrillic" style="italic" weight="normal"/>
  </font>
  <font metrics-file="c:\\fopconfig\\arialbi.xml" kerning="yes" embed-file="arialbi.ttf">
    <font-triplet name="Cyrillic" style="italic" weight="bold"/>
  </font>
</fonts>
  ...
</fonts>
```

Note: If you do not want to embed the font in the PDF document, then do not include the embed-url attribute.

4. Stop and start the application server so that these changes take effect.

Step 5: Test Your Configuration

After you perform the listed configuration steps, you need to test that you are able to create and print a PDF that uses the correct font. To test the example, you must have a ClaimCenter implementation that supports a Russian locale.

Localizing String Labels

This topic discusses the different ways you can localize the string labels that Guidewire calls display keys and typecodes:

- ClaimCenter uses display keys for field and screen labels in the ClaimCenter interface as well as labels in Studio.
- ClaimCenter uses typecodes as the code values in a typelist. You most commonly see a typelist as a drop-down list within ClaimCenter.

This topic includes:

- “Localizing Display Keys” on page 483
- “Localizing Typecodes” on page 486
- “Exporting and Importing Localization Files” on page 487

Localizing Display Keys

Studio provides a display key editor that you use to translate display keys into different languages. You find this editor under the **Localizations** node in the **Resources** tree.

To localize a display key using the editor, select a display key from the list, then select the desired locale from the drop-down list. After you choose the locale, you can enter the localized string directly into the text editor. *You must define multiple locales for this feature to be available to you.*

You can also do the following:

- Export these resources to a properties file.
- Make your translations in that file
- Import the translated resources back into Studio to associate with a specific locale, as a sort of bulk process.

Note: It is also possible to localize individual workflow step names within Studio. However, these step names are not display keys. Studio stores the workflow step names with the workflow object itself. For information on localizing workflow step names, see “Localizing Guidewire Workflow” on page 497.

ClaimCenter stores display keys that translate into U.S. standard English in the following locations:

`ClaimCenter/modules/cc/config/locale/en_US`
`ClaimCenter/modules/p1/config/locale/en_US`

If you create a new application locale, then you need to create a folder for that locale in the following directory:

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

Within this directory, you must have a folder named with the correct language and country code. This folder must contain—at the minimum—a `display.properties` file, even if that file is initially blank. It is this file that stores the application display keys. This ensures that Studio displays each locale properly as you create localized display keys.

See also

- “Display Key Localization Files” on page 484
- “Working with Missing Display Keys” on page 484
- “Different Ways to Localize Display Keys” on page 485
- “Exporting and Importing Localization Files” on page 487
- “Adding a New Locale” on page 468

Display Key Localization Files

Within Guidewire ClaimCenter, the application stores display keys in several different files. Each file contains display keys used by different parts of the application. It is possible to localize some, or all, of these files independently of each other. The display key files include the following:

File	Contains
<code>display.properties</code>	Application field labels. These show only in the ClaimCenter application interface.
<code>studio.display.properties</code>	Studio field labels. These show only in the Studio interface,
<code>gosu.display.properties</code>	Gosu messages that provide error and warning information. These show both within Guidewire Studio and within the ClaimCenter application interface.

You must have—at the very least—one `display.properties` file for each locale that you define. However, to successfully localize the Studio interface, you must also have a `studio.display.properties` file as well.

IMPORTANT You must create and store these files in the correct location manually. Studio does not create these files for you.

Place all your display key property files in the following location:

`ClaimCenter/modules/configuration/config/locale/1ocale`

To view a list of valid locale codes, refer to following web site:

http://www.loc.gov/standards/iso639-2/php/English_list.php

IMPORTANT If you are upgrading your application, be sure to review the related ClaimCenter release notes. It is possible for Guidewire to add additional display keys as part of the application upgrade. If so, you must add the new display keys to the `display.properties` files (including those for Studio and Gosu) before you localize these files.

Working with Missing Display Keys

ClaimCenter initializes the display key system as it scans all the `config/locale` directories of all modules for display key property files. Using these files, ClaimCenter generates a master list of display keys. ClaimCenter uses these values for type information, not for the actual localized display value.

For each property file, ClaimCenter loads the display keys and adds each display key to the master list under the following circumstances:

1. The master list does not already contain the display key.
2. The master list already contains the display key, but, the display key in the master list has fewer arguments than the one to add. If this is the case, then ClaimCenter logs a warning message noting that it found a display key value with different arguments in different locales. For example:

```
Configuration Display key found with different argument lists across locales: Validator.Phone
```

As ClaimCenter creates the master display key list, it scans the application locales in the following order:

1. The application default locale (as set by configuration parameter `DefaultApplicationLocale`)
2. All other locales configured for use by the server
3. The Guidewire default locale (`en_US`)
4. All remaining locales

After ClaimCenter creates the master list of display keys, it checks the display keys for the default locale against the master list. ClaimCenter then logs as errors any display keys that are in the master list but missing from the default application locale. For example:

```
ERROR Default application locale (en_US) missing display key: Example.Display.Key
```

As Studio returns the display key name, you can use that name to generate a display key value in the correct locale.

The Display Key Difference Utility Tool

Guidewire provides a display key difference tool that does the following:

- It compares each locale configured on the server against the master display key list.
- It generates a file that contains a list of any missing keys.

To generate a display key difference report, run the following command from the application bin directory:

```
gwcc displaykey-diff
```

The output follows the exact same format as the configured locales. For example, if you have three locales defined—`en_US`, `ja_JP`, and `fr_FR`—then the difference tool creates the following three directories and populates each one with a `display.properties` report:

```
ClaimCenter/build/cc/missing-display-keys/config/locale/en_US/display.properties  
ClaimCenter/build/cc/missing-display-keys/config/locale/fr_FR/display.properties  
ClaimCenter/build/cc/missing-display-keys/config/locale/ja_JP/display.properties
```

The contents of each locale file is a list of display keys that are in the master list but not in that locale. The format of the file is exactly the same as the display key configuration files. For example, the following code illustrates the contents of the file for the `en_US` locale:

```
#  
# Missing display keys for locale  
# en_US  
#  
Web.QA.I18N.ContactDetail.AddressDetail.City = Suburb  
Web.QA.I18N.ContactDetail.AddressDetail.ZipCode = Postcode
```

Note: ClaimCenter does not generate a `display.properties` file for a locale that does not have any missing display keys.

Different Ways to Localize Display Keys

There are multiple ways to localize display keys:

- Localizing Display Keys Using the Display Keys Editor
- Localizing Display Keys Using the PCF Editor

- Localizing Display Keys Using Code Completion

Localizing Display Keys Using the Display Keys Editor

It is possible to enter a localized version of a display key directly in the Display Keys editor. To access the editor, expand the **Localizations** node in the **Resources** tree.

To localize a display key using the editor, select a display key from the list, then select the desired locale from the drop-down list. After you choose the locale, you can enter the localized string directly into the text editor. It is necessary to define multiple locales for this feature to be available to you.

Localizing Display Keys Using the PCF Editor

You can also use the **Translations** tab of the PCF editor to create localized version of a display key in use on a particular PCF page. The **Translations** tab shows a list of all display keys used in the PCF file:

- If you click a display key in the list, Studio selects (outlines) the widget on the canvas that uses that display key.
- If you double-click a display key in the list, Studio opens an **Edit Display Key** dialog that shows the value of the display key in each locale defined within ClaimCenter. You can use this dialog to create locale-specific display keys that are specific to the context in which they exist.

See also

- “Using the PCF Editor” on page 337

Localizing Display Keys Using Code Completion

You can enter localized text using a Studio dialog as you create a display key directly within the Gosu editor. As you type a text string in the Gosu editor, Studio suggests that you create a display key from the text string. Pressing ALT-ENTER opens the **Create Display Key** dialog. Use the text entry field in this dialog to enter a localized string for the text for which you want to create a display key.

If you want to enter multiple entries for multiple locales, then select the **Specify values for each locale** checkbox. You can enter a localized string for each locale that exists in Guidewire ClaimCenter.

Localizing Typecodes

You use the Typelist Localization editor to facilitate typelist localization. Guidewire locates the editor under the **Localizations** node in the **Resources** tree. The Typelist Localizations editor is distinct from the Typelist editor, in which you define a typelist and its typecodes.

Note: Guidewire manages LOB typelists in a special Lines of Business editor in Studio. However, you use the Typelist Localization editor to localize the LOB typecodes as you do with any other typelist.

The Typelist Localization editor presents typelist and typecodes in a tree similar to the existing Entity Names editor. After you select a typelist—and then a typecode—from the tree, the editor presents you with an editable table of localized names and descriptions for that typecode, for each locale.

Guidewire stores typelist typekey localizations in `typelist.properties` files, one file for each locale. For example, Guidewire stores the English (US) typelist localizations in file:

```
.../modules/configuration/config/locale/en_US/typelist.properties
```

Studio stores typelist typekey localizations using the following format for name and description respectively:

```
TypeKey.typelist.typekey = localized_name  
TypeKeyDescription.typelist.typekey = localized_description
```

You use the Typelist Localizations editor to associate a locale with a typecode. By doing so, you can define a translation string for both the typecode name and description. The name and description that you set in the typelist localization editor map to the same fields on the **Codes** tab of the Typelists editor.

Note: A *typecode* is a value in a ClaimCenter drop-down list or *typelist*. See Working with Typelists for information on typelists, typecodes, and typekeys and how to work with them in Guidewire Studio.

To create a localized version of a typecode

1. Open Guidewire Studio and navigate to **Localizations** → **Typelist Localization**.
2. Select a typecode from the typelist tree.
3. Enter the translated strings for typecode name and description in the appropriate fields for each locale.

Note: If the **Typelist Localizations** editor does not contain the locale that you want to select, then that locale does not yet exist in Studio. To add additional locales to ClaimCenter, follow the instructions in “Adding a New Locale” on page 468.

See also

- For details of how to export a properties file and then import the translated file back into Studio, see “Exporting and Importing Localization Files” on page 487.

Accessing Localized Typekeys from Gosu

Gosu provides three String representations that you can use for typekeys. The following list describes them.

Typekey property	Description
<code>typekey.Code</code>	String that represents the typecode
<code>typekey.DisplayName</code>	Localized language version of the entity name
<code>typekey.UnlocalizedName</code>	Name listed in the data model

For example, to extract (localized) entity name information, you can use the following:

```
var displayString = myCode.DisplayName
```

Or, as a concrete example:

```
print(AddressType.TC_BUSINESS.DisplayName)
```

Exporting and Importing Localization Files

It is possible to export all display key and typecode resources to an external file and make your translations in that file. You can then import the display keys back into Studio to associate with a specific locale.

To assist you in this process, Guidewire provides localization commands on the contextual right-click menu. The following menu items appear if you select either **Display Keys** or **Typelists** from the **Resources** tree.

Export Translation File	Exports the <code>display.properties</code> file—and all the typekeys or typelists—for the locale of your choice. If you have not created additional locales, this command is inactive. By default, Studio places the export file in the root of the installation directory. It is possible to change the file save location, however.
Import Translation File	Imports the <code>display.properties</code> file for the locale of your choice. You typically use this option to import a translated <code>display.properties</code> file for the bulk import of translated display keys and typecodes. If you have a number of typecodes for which you want translated names and descriptions, then you can use this option rather than entering them individually. Guidewire also provides a command line tool that you can use to manually import a translation file into Studio.

Note: The translation import/export commands are only active if multiple locales exist within ClaimCenter. You must define multiple locales for these commands to be active.

To translate a display properties file

1. To export a translation file from Studio, right-click **Localizations** in the **Resources** tree, click **Export translation file**, and then select the source and target locales.

The options available in this submenu depend on the locales you define. For example, if you define English, French, and German locales, then you see the following six combinations:

- *English (US) to French (CA)*
- *English (US) to German (DE)*
- *French (CA) to English (US)*
- *French (CA) to German (DE)*
- *German (DE) to English (US)*
- *German (DE) to French (CA)*

2. Save the file as prompted.

Studio copies all entries in the source locale `display.properties` file into the export file and merges the source locale entries with any entries contained in the target locale `display.properties` file. If the target locale `display.properties` file already contains entries, the export file groups these together and labels them as translated. The file groups the remaining displays together and marks them as not translated.

3. Translate the file contents into the target language. You can translate the values, but do not modify the display key names.

4. After completing the translation, import the file back into Studio. Right-click **Localizations**, then click **Import translation file**.

This action opens a list of target locales from which to choose. After you select a locale, Studio overwrites any existing `display.properties` file in the target locale directory, replacing it with the translated version.

To import a translation file manually

Guidewire provides a command line tool that you can use to manually import a translation file into Studio. To use this tool, do the following:

1. Navigate to your installation `bin` directory, for example:

`ClaimCenter/bin`

2. Run the following command:

```
gwcc import-l10ns -Dimport.file="translation_file" -Dimport.locale=destination_locale
```

The command takes the following parameters:

- | | |
|-----------------------------|--|
| <code>Dimport.file</code> | Specifies the file that contains the translations. It must be in the same format as an export file from Studio. By default, Studio places the export file in the root of the installation directory: <ul style="list-style-type: none">• If you leave the import translation file in the same location, then you need enter only the name of the file to import.• If you move the translation file to a different location, then you enter an absolute path or a relative path to the file from the root of the installation directory. |
| <code>Dimport.locale</code> | Specifies the destination locale for the translations. The locale must match a ClaimCenter locale that you have defined through Studio, for example: <code>fr_FR</code> or <code>ja_JP</code> . |

Localizing the Development Environment

This topic covers how you localize the Guidewire development environment, which includes localizing the Guidewire Studio interface.

This topic includes:

- “Viewing Double-byte Characters in Studio” on page 491
- “Changing the Studio Locale” on page 492
- “Localizing Rule Set Names and Descriptions” on page 492
- “Setting a Locale for a Gosu Code Block” on page 493
- “Localizing Gosu Error Messages” on page 494
- “Localizing Administration Tool Argument Descriptions” on page 494
- “Localizing Logging Messages” on page 496

Viewing Double-byte Characters in Studio

To correctly view double-byte characters within Guidewire Studio, you most likely need to set your Microsoft Windows operating system to recognize double-byte characters. (You might do this, for example, if you set the localization to an East Asian language such as Japanese.) Otherwise, Studio displays a series of blocks (hollow squares) in place of the character representation.

Guidewire Studio reads Windows standard font settings for menus and other items. This is usually a Windows standard font, such as Tahoma. For details, on how the Windows operating system handles fonts, see the following Microsoft site:

http://www.microsoft.com/globaldev/getwr/steps/wrg_font.mspx

As Microsoft states (on this site), the Windows standard fonts do not support East Asian languages, meaning double-byte characters. To see these characters properly in Guidewire Studio, perform the following steps within the Microsoft Windows operating system.

To set double-byte character representations in Microsoft Windows

1. Within Microsoft Windows, navigate to **Control Panel** → **Regional and Language Options** and set the following:

Tab	Drop-down field	Set to
Regional Options	Standards and Formats	The double-byte language (Japanese, for example)
Advanced	Language for non-Unicode programs	The double-byte language

2. Save your work. Windows prompts you to restart. You must do so before continuing.

3. Within Microsoft Windows, navigate to **Control Panel** → **Display** and set the following:

Tab	Item drop-down (Advanced)	Set to
Appearance	Message Box	An East Asian friendly font. (for example, MS Gothic)

At this point, it is possible to set up a valid double-byte language locale and have Studio display it properly.

Additional Information

The following Microsoft sites may also be of interest:

- **Enabling International Support in Windows XP/Server 2003 Family.** <http://www.microsoft.com/globaldev/handson/user/xpintlupp.mspx>
- **NLS Terminology.** [http://msdn.microsoft.com/en-us/library/ms776246\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms776246(VS.85).aspx)

Changing the Studio Locale

After you add one, or more, additional locales, Studio displays a locale selector in the bottom right-hand corner of the screen. The locale selector displays a drop-down list of all the locales defined within Guidewire ClaimCenter.

Selecting a different locale changes all user-visible text within Studio to that locale, as specified in the `studio.display.properties` file for that locale. (This file contains the Studio field labels.)

Note: To add a locale see “Adding a New Locale” on page 468.

Localizing Rule Set Names and Descriptions

To display a rule set name and description in another language, you can translate these items in the definition file for that rule. These files end in `*.grs`. Within a `.grs` file, you can translate the following:

```
@gw.rules.RuleName("Translated String")
@gw.rules.RuleSetDescription("Translated String")
```

Guidewire stores these files in the following directories:

```
ClaimCenter/modules/cc/config/rules/...
ClaimCenter/modules/pl/config/rules/...
```

You must create your own translated version of a `.grs` file and place it in the following directory:

```
ClaimCenter/modules/configuration/config/rules/...
```

Again, only modify files in the `modules/configuration/config` directory. Otherwise, you can negate your installation of Guidewire ClaimCenter.

Note: You can only display a single translated version of a rule set name or description. Changing the user locale for Studio does not affect these items.

Setting a Locale for a Gosu Code Block

It is possible to set a specific locale in any Gosu block (within any Gosu code) by wrapping the Gosu code in the following method.

```
gw.api.util.LocaleUtil.runAsCurrentLocale(alternateLocale, \ -> { code })
```

These parameters take the following form:

Parameter	Description
alternateLocale	The locale to use in the form of an <code>ILocale</code> , for example: <code>gw.i18n.ILocale.fr_FR</code>
\ -> { code }	A runnable Gosu block—the Gosu code to run in a different locale

Example 1. You can use this method to localize Gosu code that you add to any workflow step, in an `Enter Script` block. For example, in PolicyCenter, you can use an `Enter` script Gosu block on the `IssueCancellation` step to issue the cancellation documents in the primary language of the account holder. This locale can be different from the default application locale or any locale set for the workflow in general. For example:

```
uses gw.api.util.LocaleUtil  
LocaleUtil.runAsCurrentLocale( LocaleUtil.toLocale( PolicyPeriod.Policy.Account.PrimaryLanguage ),  
    \ -> PolicyPeriod.CancellationProcess.issueCancellation() )
```

The Gosu block uses the `LocaleUtil.toLocale` method. You use this method to translate the first parameter expression (`PolicyPeriod.Account.PrimaryLanguage`) into the necessary `ILocale` type.

Example 2. In a more generic example, suppose that you want to format a date for a particular language. For example, you want to represent a date in the format `yyyy-dd-MM` (as is typical in the French language). You can use `SimpleDateFormat` with a `Date` object and call the `runAsCurrentLocale` method to format the date for the French locale. For example:

```
uses gw.api.util.LocaleUtil  
uses gw.i18n.ILocale  
uses java.util.Date  
uses gw.api.util.DateUtil  
  
var formattedDate : String  
LocaleUtil.runAsCurrentLocale(ILocale.FR_FR,  
    \ -> { formattedDate = DateUtil.currentTimeMillis().format("long") })
```

Additional Useful Methods on `gw.api.util.LocaleUtil`

In addition to the `runAsCurrentLocale` method, `gw.api.util.LocaleUtil` contains other methods useful in working with localization. Some of the more important ones are:

Method	Description
canSwitchLanguage	Returns true if the current user is allowed to switch to a different language.
getAvailableLanguages	Returns a list of all available languages as defined in the <code>LanguageType</code> typelist.
getCurrentLocale	Returns the current locale, which can be based on the user language setting or on the browser setting.
getCurrentUserLanguage	Returns the language set for the current user.
getDefaultLanguage	Returns the language associated with the default locale (as set by configuration parameter <code>DefaultApplicationLocale</code>).

Method	Description
getDefaultLocale	Return the applications default locale as set by configuration parameter DefaultApplicationLocale.
toLanguage	Given a locale, return a language type.
toLocale	Given a language type, return a locale.

Localizing Gosu Error Messages

Guidewire ClaimCenter stores the text strings that Gosu errors generate in the following location:

ClaimCenter/modules/p1/config/locale/en_US/gosu.display.properties

For example, file *gosu.display.properties* provides the following English language assignments:

```
ARRAY = Array
BEAN = Bean
BOOLEAN = Boolean
DATETIME = DateTime
FUNCTION = Function
IDENTIFIER = Identifier
METATYPENAME = Type
NULLTYPENAME = Null
NUMERIC = Number
OBJECT_LITERAL = ObjectLiteral
STRING = String
MSG_BAD_IDENTIFIER_NAME = Could not resolve symbol for : {0}
...
...
```

It is possible to create translated versions of the error message text strings. Guidewire then shows the translated or localized version of the error message instead of the default English version if you set Studio (or ClaimCenter) to that locale. To do this, you follow a similar procedure to that you performed if you translated the *display.properties* file.

To generate localized Gosu error messages

1. Within the application file structure, navigate to the following location:

ClaimCenter/modules/p1/config/locale/en_US

2. Make of copy of the *gosu.display.properties* file in that directory and place the copy in the following location:

ClaimCenter/modules/configuration/config/locale/<locale_folder>

Note: If a directory folder does not exist for the chosen locale, then first perform the steps in “Adding a New Locale” on page 468.

3. Translate the text strings as required.

To see translated error strings for a particular locale, you must set Studio (or ClaimCenter) to that locale first.

See Also

- “Changing the Studio Locale” on page 492
- “Setting the User Locale” on page 476

Localizing Administration Tool Argument Descriptions

Guidewire provides a set of administration tools defined in Gosu. You use these tools, for example, to control server behavior, to load data, and to generate tools for use in the Guidewire configuration environment.

These command line tools derive their arguments from properties on a class that ClaimCenter initializes at the start of each tool, depending on the tool. ClaimCenter stores these files in the following location:

ClaimCenter/admin/src/gw/cmdline/util/...

In the base configuration, this directory contains the following command line argument files:

- ImportToolArgs.gs
- SystemToolArgs.gs
- TableImportArgs.gs
- WorkflowToolsArgs.gs
- ZoneImportToolArgs.gs

Each of these Gosu classes contains a set of static properties with associated comments. By default, ClaimCenter derives the description of each command line argument from the comments in the file associated with that administration tool. However, it is possible to create a separate property file, with localized name/value pairs of the property descriptions and associate that file with a tool. ClaimCenter then reads the associated property file and displays localized versions of the tool argument descriptions.

For example, in `SystemToolsArgs`, you see the following definition for the `server` command line argument:

```
/**  
 * The password to use  
 */  
@ShortName( "password" )  
@LongName( null )  
static var _password : String as Password
```

Suppose that you want to localize the description of the `Password` argument for a French (France) locale. You need to create the following file:

`ClaimCenter/admin/src/gw/cmdline/util/SystemToolArgs_fr_FR.properties`

and populate it with the following:

```
Password=Le mot de passe à utiliser
```

In this way, it is possible to localize the property name descriptions (which are the command line argument descriptions) in these files. To do so, you need to do the following:

- Create a properties file for the locale and populate it with translated name/value pairs.
- Name the file using the standard Java convention for differentiating between various localizations.
- Place the properties file in the same directory as the class file that it modifies.

IMPORTANT Guidewire ClaimCenter displays the localized descriptions for the administration tools for the *operating system* locale, **not** the locale set for Guidewire ClaimCenter.

To localize the command line argument descriptions

1. Create a properly translated properties file, with a name/value pair for each property in the class for which you are providing localization.

- *Name* is the property in class `system_tools`, for example, `SystemToolsArgs.Server`.
- *Value* is the properly localized description of that property name.

The property file must have a name/value pair for each property in the class for which you want to provide localization. If there is no entry for a given property, ClaimCenter uses the default documentation from the comment on the property in class.

2. Name the property file with the *same name* as the tool class, using the standard Java conventions for differentiating between various localizations. For example, if creating a localized `SystemToolArgs` properties file for German, use the following:

`SystemToolsArgs_de_DE.properties`

3. Place the translated properties file in the *same directory* as the tool class, which is:

`ClaimCenter/admin/src/gw/cmdline/util/...`

To access the command line tools in a chosen locale

Guidewire ClaimCenter displays the localized descriptions for the command line tools for the *operating system* locale, **not** the locale set for Guidewire ClaimCenter. Thus, to run the command line tools in a specific locale, do one of the following:

- Set your operating system to the chosen locale.
- Set the command line window to the chosen locale.

The following steps illustrate how to set the locale of a command line window using the GOSU_OPTS command.

1. Open a command line window and navigate to the following directory:

ClaimCenter/admin/bin

2. Enter the following command, replacing <locale> with the exact same locale that you used to name your localization property file.

ClaimCenter/admin/bin/set GOSU_OPTS=-Duser.language=<locale>

For example, if you create a property localization file named **SystemToolsArgs_de_DE.properties**, then enter the following:

ClaimCenter/admin/bin/set GOSU_OPTS=-Duser.language=de_DE

3. Run your administration tool.

You can view a list of arguments and descriptions by entering the tool name without any arguments. For example, if you enter the following, ClaimCenter displays a list of the arguments that you can use with the system tools command:

ClaimCenter/admin/bin/system_tools

See Also

- “Generating Java and SOAP API Libraries” on page 64 in the *Installation Guide*
- “ClaimCenter Administrative Commands” on page 169 in the *System Administration Guide*

Localizing Logging Messages

As described—in Localizing Gosu Error Messages—it is possible to localize Gosu error messages. (Guidewire provides access to the **gosu.display.properties** file, which you can localize.) It is also possible for you to localize any error messages that you create yourself through Studio. This can a message embedded in a Gosu rule or Gosu class that prints information to the Studio console.

There are two instances, however, in which Guidewire does not provide localized messages:

- Java stack traces
- Apache log4j messages

Java stack traces. Guidewire does not display the originating Java error message to the ClaimCenter user in the event that ClaimCenter throws an error or exception. Instead, Guidewire logs the Java stack in the Guidewire ClaimCenter log files and typically shows the user a more generic message that you can localize. For example:

There has been an error on the server. Contact your system administrator.

In any case, Java does not provide localized stack traces. It simply lists the package, class, and line number of the location of the error or exception in the code.

Apache log4j messages. Guidewire does not provide localized translations of log4j messages in the base ClaimCenter configuration.

Localizing Guidewire Workflow

This topic discusses localization as it relates to Guidewire workflow.

This topic includes:

- “Localize a Workflow” on page 497
- “Localizing Gosu Code in a Workflow Step” on page 499

Localize a Workflow

Before advancing a workflow, the Workflow engine determines the *locale* in which to execute the workflow. This is the locale that ClaimCenter uses for display keys, dates, numbers, and other similar items. Unless set otherwise, this is the application default locale.

This topic covers the following:

- Setting a Locale for a Workflow
- Localizing Workflow Step Names
- Creating a Locale-Specific SubFlow

Setting a Locale for a Workflow

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. (See “Localizing Templates” on page 517 for how you can localize application documents, notes, and emails.) You can, however, set a workflow locale that is different from the default application locale.

To set a locale on a workflow, click in the background area in the workflow Layout view. This opens the properties area at the bottom of the workflow area. You can enter either a fixed name for the locale or use a Gosu expression that evaluates to a valid `ILocale` type to retrieve a locale. For example:

Type	Gosu
Fixed string	<code>gw.i18n.ILocale.fr_FR</code>

Type	Gosu
Variable expression	<pre>gw.api.util.LocaleUtil.toLocale(thisClaim.Claimant.PrimaryLanguage) gw.api.util.LocaleUtil.toLocale(Group.Supervisor.Contact.PrimaryLanguage)</pre>

Localizing Workflow Step Names

Guidewire ClaimCenter stores the names of the workflow steps with workflow object itself. The names are not strings that you can turn into display keys, which you can then translate. Instead, ClaimCenter provides a different mechanism to enable the localization of workflow step names.

You can display a translated workflow step name by first selecting the workflow step, then clicking on the **Step Name Localizations** tab. (This tab appears in the properties area at the bottom of the workflow editor.) Click **Add** and enter the translated step name. After you make the change, select the locale from the Studio locale drop-down list in the bottom right-hand corner of the screen. Studio then displays the translated name for that locale in the workflow editor.

If you localize a step name, ClaimCenter displays the localized version of the step name in the following locations:

- Workflow steps in Studio
- Workflow summary in ClaimCenter on the **Find Workflows** search screen. (You must use an administrative account to see this screen.) The summary shows the last completed step. If localized in Studio, the step shows in that form on this ClaimCenter screen. This happens *only if you set the locale* using the ClaimCenter language selector (in the right-hand corner of the ClaimCenter screen).
- Workflow log in ClaimCenter on the **Workflow Detail** screen. (You access this screen by clicking the workflow name on the **Workflows Search** screen.) The log shows all workflow steps. If localized in Studio, these steps show in that form on this ClaimCenter screen. This happens *only if you set the locale* using the ClaimCenter language selector (in the right-hand corner of the ClaimCenter screen).

Creating a Locale-Specific SubFlow

As described in “Workflow Subflows” on page 450, you can create a child workflow (a subflow) in Gosu using the following methods on **Workflow**. Each method handles the locale of the subflow differently.

Method	Description
<code>createSubFlow</code>	Creates a child subflow <i>synchronously</i> , meaning that the Workflow engine starts the subflow immediately upon method invocation. The new subflow automatically uses the <i>default application locale</i> , not the locale of the parent workflow. Thus, if you set the locale of the parent workflow to be different from the default application locale, the subflow does not inherit that locale.
<code>createSubFlowAsynchronously</code>	Creates a child subflow <i>asynchronously</i> , meaning that the Workflow engine does not start the subflow until it finishes executing all code in the Gosu initialization block. Again, the subflow uses the default application locale, not the locale set for the workflow itself. As the Workflow engine executes all the Gosu code in the block before starting the subflow, it is possible to set the locale of the subflow before the workflow starts.
<code>instantiateSubFlow</code>	Creates a child subflow, but does not start it. If desired, you can modify the subflow that the method returns before you start the subflow.

To create a subflow that inherits the locale of the parent workflow

1. Define a workflow that has the `LanguageType` property. See “Creating New Workflows” on page 440 for information on how to create a new workflow with a `LanguageType` property.
2. Set the locale for this subflow so that it uses your desired language. See “Setting a Locale for a Workflow” on page 497 for details.

3. Instantiate the subworkflow using the `instantiateSubFlow` method rather than the `createSubFlow` method.
4. Set the `LanguageType` property on the instantiated subflow to the locale of the parent workflow.
5. Start the subflow, using one of the workflow start methods described at “Instantiating a Workflow” on page 444.

Localizing Gosu Code in a Workflow Step

It is possible to localize Gosu code that you add to any workflow step (an `Enter Script` block, for example). To do so, wrap the Gosu code in the following method.

```
gw.api.util.LocaleUtil.runAsCurrentLocale(alternateLocale, \ -> { code } )
```

See “Setting a Locale for a Gosu Code Block” on page 493 for examples of how to work with this method.

Localizing Shared Administration Data

This topic discusses how to localize shared administration data through the use of a localization database column. For example, all users in the ClaimCenter application share activity patterns. (ClaimCenter uses activity patterns to create new activities.) Through the use of a localization column, users in different locales can see an activity name appropriate for their preferred locale.

This topic includes:

- “Shared Administration Data” on page 501
- “Localized Entities” on page 502
- “Localization Database Tables” on page 503
- “Localized PCF Files” on page 503

Shared Administration Data

Guidewire ClaimCenter provides mechanisms for localizing the string labels of fields and typelist values.

- For interface field labels, ClaimCenter uses various `display.properties` files to store localization information.
- For typelists, ClaimCenter stores localization information directly with the typelist files themselves.

However, there is another class of data that ClaimCenter uses mainly for administration and reference, which ClaimCenter stores directly in the application database as values. This type of data includes the following:

- Activity patterns
- Groups
- Regions

Other types of administrative data that ClaimCenter stores in the database include the following:

- Catastrophes

- Geographical data (city, province, and country names, for example)
- Holidays
- Values for industry and other organization codes

Localized Entities

To accommodate multilingual users who want to use shared administration data, Guidewire provides a mechanism to create a database column on an entity that stores localization information. To accomplish this, Guidewire provides a `<localization>` XML child element to `<column>` in the metadata. The existence of this node triggers a localized column within Guidewire ClaimCenter. The `<localization>` subelement has one attribute, `tableName`, which is the table name of the localization *join* table.

IMPORTANT Do not exceed a `tableName` length of 16 characters. If you do, the application server refuses to start.

The `<localization>` subelement takes the following form:

```
<?xml version="1.0"?>
<entity ... >
  ...
  <column ... >
    <localization tableName="..."/>
  </column>
  ...
</entity>
```

For example, in the base configuration, Guidewire localizes the `Description` column on the `ActivityPattern` entity. This means that this column can include localized values for each language (locale) that you define in the `LanguageType` type list.

The `ActivityPattern` entity looks similar to the following:

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
  admin="true"
  desc="An activity pattern is a template for an activity. "
  entity="ActivityPattern"
  exportable="true"
  extendable="true"
  javaClass="com.guidewire.pl.domain.activity.ActivityPatternBase"
  platform="true"
  table="activitypattern"
  type="retireable">
  <implementsEntity name="DecentralizedEntity"/>
  <column desc="Description of the activity pattern." name="Description" type="mediumtext">
    <localization tableName="actpat_desc_110n"/>
  </column>
  ...
</entity>
```

A Localized Catastrophe Entity Example

The following ClaimCenter `Catastrophe` entity code provides another example of localizing certain columns on an entity, in this case the `Name` and `Description` fields.

```
<?xml version="1.0"?>
<entity admin="true"
  desc="Catastrophe"
  entity="Catastrophe"
  exportable="true"
  extendable="true"
  javaClass="com.guidewire.cc.domain.catastrophe.Catastrophe"
  platform="false"
  table="catastrophe"
  type="retireable">
  <fulldescription>Represents a catastrophe.</fulldescription>
  ...
  <column desc="Description of the catastrophe." name="Description" type="shorttext">
    <localization tableName="cat_desc_110n"/>
  </column>
</entity>
```

```
</column>
<column desc="Name of the catastrophe." name="Name" type="varchar">
  <columnParam name="size" value="60"/>
  <localization tableName="cat_name_l10n"/>
</column>
...
</entity>
```

Localization Database Tables

ClaimCenter stores the localized values in the join table that it auto-generates for each localized column that you create. As you define a localized column (see “Localized Entities” on page 502), Guidewire recommends that you use the following format for the join table name. *You must also ensure that the complete table name length is less than 16 characters:*

```
<MainEntityNameAbbreviation>_<ColumnNameAbbreviation>_L10N
```

Thus, to create a table name, you use the main entity name and the name of the column that you wish to localize. For example:

```
ActPtrn_Sbj_L10n
```

Join table columns. The auto-generated join table includes the following columns (among others):

- An Owner column, the type of which is an integer that represents an ID of the owner.
- A Language column, the type of which is the `LanguageType` typelist.
- A Value column, the type of which matches the type of the corresponding column in the main table. This column contains the actual localized value.

It is important to understand the following:

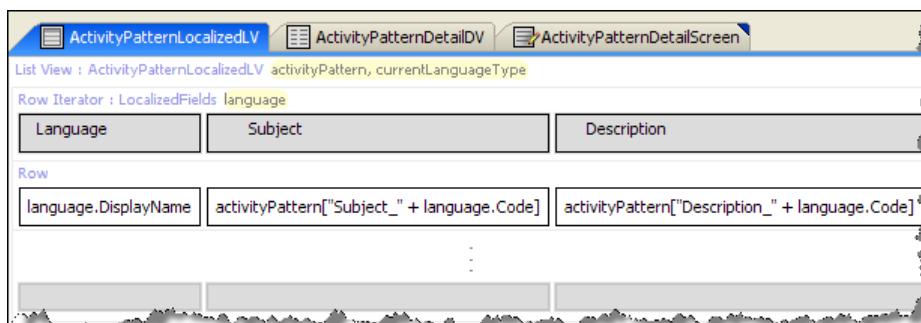
- ClaimCenter stores the localization for the *default* locale as a column on the main entity.
- ClaimCenter stores the localizations for all other locales as columns in the join table.

Localized PCF Files

After you add a localization column to an entity, you need to expose this functionality through the application interface, within a PCF file. To use the activity pattern example again, the `ActivityPattern` code defines the activity pattern subject and description as localized:

```
<column desc="Description of the activity pattern." name="Description" type="mediumtext">
  <localization tableName="actpat_desc_l10n"/>
</column>
<column desc="Subject field of the activity." name="Subject" type="shorttext">
  <localization tableName="actpat_subj_l10n"/>
</column>
```

In the base configuration, Guidewire provides this functionality by embedding an `ActivityPatternLocalizedLV` PCF within the `ActivityPatternDetailScreen`. The following graphic illustrates the `ActivityPatternLocalizedLV` PCF file (in the Studio PCF editor).



The following graphic illustrates the **ActivityPatternDetailScreen**. (Depending on your Guidewire application, this screen can have different fields and be slightly different.)

This screenshot shows the configuration of the **ActivityPatternDetailScreen**. The top section contains various mandatory fields: **Mandatory *** (ActivityPattern.Mandatory), **Code** (ActivityPattern.Code), **Recurring** (ActivityPattern.Recurring), **Subject *** (ActivityPattern.Subject), **Priority** (ActivityPattern.Priority), **Type *** (ActivityType.TC_GENERAL), and **Pattern Level *** (ActivityPattern.PatternLevel). To the right of these are **Escalation Include Days** (ActivityPattern.EscalationInclDays) and **Target Include Days** (ActivityPattern.TargetIncludeDays). Below these are sections for **Templates**, specifically **Document Template** (ActivityPattern.DocumentTemplate) and **Email Template** (ActivityPattern.EmailTemplate). A panel at the bottom, titled "Panel Ref", contains a "List View" for **ActivityPatternLocalizedLV** with fields for **language** (Language, Subject, Description) and a "Row" iterator (language) with columns for language.DisplayName, activityPattern["Subject_" + language.Code], and activityPattern["Description_" + language.Code].

The base ClaimCenter configuration contains, for example, an **Administration** → **Catastrophes** → **New Catastrophe** screen in which you can enter localized values for the name and description of the catastrophe. The **CatastropheLocalizedLV** defines the localization area at the bottom of the **New Catastrophe** screen. (This file only exists in the base configuration in ClaimCenter.)

This screenshot shows the configuration of the **CatastropheDetailScreen**. It includes a "List View" for **CatastropheLocalizedLV** with fields for **language** (Language, Name, Description) and a "Row" iterator (language) with columns for language.DisplayName, catastrophe["Name_" + language.Code], and catastrophe["Description_" + language.Code].

In the base configuration, ClaimCenter embeds **CatastropheLocalizedLV** within the **CatastropheDetailScreen**.

The following graphic illustrates the base ClaimCenter New Catastrophe screen.

New Catastrophe (Up to Catastrophes)

Catastrophe

Name	*	<input type="text"/>
Description	*	<input type="text"/>
CAT No	*	<input type="text"/>
PCS Serial No		<input type="text"/>
Type	*	<none selected> <input type="button" value="▼"/>
Begin Date	*	<input type="text"/> .../.../.... <input type="button" value="..."/>
End Date	*	<input type="text"/> .../.../.... <input type="button" value="..."/>

Areas Covered

Zone Type <none selected>

Coverage Peril(s)

Add Remove

* Loss Type	* Peril(s)	Comments
-------------	------------	----------

Localization

Language	Name	Description
English (US)		

Localizing Field Validators

This topic discusses how to create field validation that is country-specific, meaning that the input validation mask changes depending on the country associated with the field. This is useful if you want to support data from multiple countries as the validation and input masks differ for each country.

This topic includes:

- “Localizing Field Validation” on page 507
- “Configuring Localized Error Messages for Field Validators” on page 508
- “Validating Country-Specific Entity Fields” on page 509

Localizing Field Validation

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.

There are several different ways to approach the localization of field validators. You can, for example:

- Use one set of field validators for all countries, but create translated error messages for each locale.
- Use multiple sets of field validators, which provides the ability to create localized validators by country.

In the base configuration, Studio stores field validator information in the **Resources → Data Model Extensions** folder, in `fieldvalidators.xml`.

Using a Single Field Validators File

If you are using one set of field validators for all countries, you use the base configuration `fieldvalidators.xml` file in **Data Model Extensions** to define your validators. Within this file, you must make the `description` field for each validator a display key. You can then provide multiple translated versions for this display key. For example, in the base configuration, file `fieldvalidators.xml` contains the following validator definition:

```
<ValidatorDef description="Validator.Phone" input-mask="###-###-### x###" name="Phone"  
value="[0-9]{3}-[0-9]{3}-[0-9]{4}( x[0-9]{0,4})?" />
```

In this case, `Validator.Phone` must be a display key for which you provide translated versions. See “Configuring Localized Error Messages for Field Validators” on page 508 for an example of how to do this.

Using Multiple Field Validator Files

It is also possible to create multiple field validation files if you want to create country-specific field validation. This type of localized field validation **only** works with fields of data type `LocalizedString`. You cannot use it, for example, with `Money` fields, or other non-`String` data types. In addition, there must also be an entity property that stores the specific country code for that entity. You set and maintain this property through code.

To implement, you create country-specific folders in **Data Model Extensions** and provide localized versions of the `fieldvalidators.xml` file for each country, along with translated versions of the validator error messages. See “Validating Country-Specific Entity Fields” on page 509 for an extended example of how this works in practice.

IMPORTANT ClaimCenter triggers field validation off the *country* associated with an entity field. It does **not** trigger off the *locale* set through the ClaimCenter interface (which is the locale of the logged-in user). For example, for an address, ClaimCenter associates an address format with the country listed for that address, **not** the locale of the user.

To create a country-specific field validators file

1. Select the `fieldvalidatorscountries` folder (in Studio Resources → **Data Model Extensions**).
2. Select **New → Other** file from the right-click menu.
3. Enter the name of the file. You must use the following name format for the file:

`fieldvalidators_<country_code>.xml`

ClaimCenter uses the name of the file to determine the country for which these field validators are valid. The `<country_code>` that you use must match an existing country code that exists in the **Country** typelist. A country code consists of a two-character uppercase ISO 3166-1-alpha-2 code for a country. For France, the two-character abbreviation is FR, for Japan, it is JP, and for Switzerland, it is CH, for example.

To view a list of valid country codes, refer to the following web site:

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

4. Enter field validators specific to this country.

IMPORTANT File `fieldvalidators.xml` contains the *default* definitions for field validators for US standard English. If multiple countries exist, then ClaimCenter uses this file as the *default* definition if you do not specify a country. ClaimCenter also uses this file as the default if a country-specific validator file does not exist for the specified country.

See Also

- “Field Validation” on page 293

Configuring Localized Error Messages for Field Validators

You can define a localized display key for each field validator format to provide a localized validation error message for a field validator of a given type. For example, the default `fieldvalidators.xml` file (for US standard English) contains a `Phone` validator for a ten digit number plus optional extension:

```
<ValidatorDef description="Validator.Phone" input-mask="###-###-### x###" name="Phone"  
value="[0-9]{3}-[0-9]{3}-[0-9]{4}( x[0-9]{0,4})?"/>
```

Thus, in the base configuration, the error message (display key) attached to `Validator.Phone` looks similar to the following:

```
{0} must be a 10-digit phone number, with an optional extension
```

To create localized field validator error messages

To create localized error messages for incorrect `Phone` input, do the following:

1. Create a separate field validator file for each country. For example, for the United States, Great Britain, and Spain, create the following field validator files. Store each in the `fieldvalidatorscountries` folder (in `Data Model Extensions`).
 - `fieldvalidators_US.xml`
 - `fieldvalidators_GB.xml`
 - `fieldvalidators_ES.xml`

2. Within each of the country field validator files, enter the appropriate validator definition. For example, for the United States, enter the following.

```
<ValidatorDef description="Validator.Phone.US" input-mask="###-###-#### x###" name="Phone"  
value="[0-9]{3}-[0-9]{3}-[0-9]{4}( x[0-9]{0,4})?"/>
```

Notice that the validator definition no longer references a `Validator.Phone` display key for the error message. Now, it references a `Validator.Phone.US` display key.

3. Create a separate display key for each country that you referenced in the field validator files. For example, create the following display keys and populate them accordingly.

- `Validator.Phone.US`
- `Validator.Phone.GB`
- `Validator.Phone.ES`

Validating Country-Specific Entity Fields

Suppose that you want to define a localized text string that has a different input mask and field validator for each country. This can be, for example, an address postal code that changes its format for each country. Or, it can be the telephone number, which again can vary by country.

Example Set Up

This basic example adds an entry field named `Some Localized String` to the `Policy Info` screen used in the PolicyCenter Submission wizard. It sets an input validation mask that changes depending on whether the country on the policy address is the United States or Great Britain. It also creates separate error messages (display keys) for incorrect user input for those two countries.

PolicyCenter looks something like this after you incorrectly enter data into the Some Localized String field. Notice that the address country is set to United States of America.

Policy Info

Some Localized String : must be three digits (United States).

Date Quote Needed: 09/03/2009

Primary Named Insured

Name: Metals Unlimited

Phone: 651-234-0000 x....

Policy Address

Change To:

0000 Bridgepointe Parkway
Floor 0000
Developer Unit Habitation Cube #00
San Mateo, CA 94404-0000

County: San Mateo

Country: **United States of America**

Address Type: Home

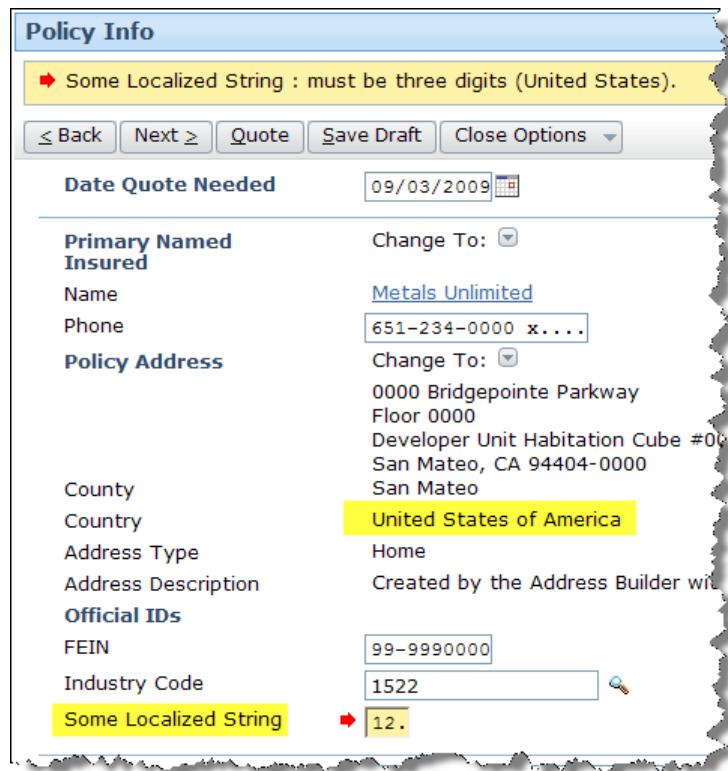
Address Description: Created by the Address Builder with...

Official IDs

FEIN: 99-9990000

Industry Code: 1522

Some Localized String: **12.**



Notice how the Some Localized String field changes its input format (and error message) if you set the address country to Great Britain.

Policy Info

Some Localized String : must be six digits (Great Britain).

Date Quote Needed: 09/03/2009

Primary Named Insured

Name: Stone Cold

Phone: 555-555-5555 x....

Policy Address

Change To:

1 Main St
Apt 1
City, AR 34546

County: County

Country: **United Kingdom of Great Britain and N. Ireland**

Address Type: Billing

Official IDs

SSN:-....

Industry Code:

Some Localized String: **123...**



Example Files

The example creates or modifies the following files or display keys:

File or display key	Purpose
AccountInfoInputSet.pcf	Adds an entry field named Some Localized String to the Policy Info page of the PolicyCenter Submission wizard.
PolicyAddress.etx	Extends the PolicyAddress entity by adding a SomeLocalizedString column of type LocalizedString and adds a validator for this column (field).
fieldvalidators.xml	Adds a default validator definition for SomeLocalizedString .
fieldvalidators_US.xml fieldvalidators_GB.xml	Adds country-specific validation for the desired countries.
Validator.Phone.US Validator.Phone.GB	Adds new display keys for incorrect user input for each country.
PolicyAddressEnhancement.gsx	Not necessary to create or modify for this example as it exists in the base configuration. Depending on the entity that you modify, however, it can be necessary to create a Gosu entity enhancement.

Example Steps

This example requires multiple steps. It illustrates these steps by extending the **PolicyAddress** entity.

1. First, you need to create an extension **PolicyAddress.etx** file to define the new localized column (in this case, **SomeLocalizedString**) and to set the associated validator and other parameters.
2. Next, you need to modify PCF file **AccountInfoInputSet** and add input field **Some Localized String**.
3. Then, you need to add the default validation to the **fieldvalidators.xml** for **SomeLocalizedString**.
4. Next, you need to create new validator files for the country-specific field validators and input masks and create the country-specific display keys to use as validation error messages.
5. Finally, you need to write a **Gosu** enhancement that you use to trigger the country-specific validation behaviour.

IMPORTANT This example does not persist data to the database. It simply illustrates the process of creating country-specific validations for any given entity field.

Step 1: Extend the PolicyAddress Entity

The first step in the process is to add a column to the **PolicyAddress** entity and associate a field validator with that column (field).

IMPORTANT You cannot use a single regular expression on a field that varies by locale, unless you use the **LocalizedString** data type for that field.

To add a column to PolicyAddress

1. Create an **PolicyAddress.etx** file if one does not already exist.
 - a. Select the **Data Model Extensions** → **extensions** folder.
 - b. Right-click, select **New** → **Other file**.
 - c. Enter the name of the file in the dialog. You must add the **.etx** file extension. **PolicyCenter** does not do this for you.

2. Enter the following in `PolicyAddress.etc`.

```
<?xml version="1.0"?>
<extension
  xmlns="http://guidewire.com/datamodel"
  entityName="PolicyAddress">
  <column desc="Some Localized String" name="SomeLocalizedString" type="localizedstring">
    <columnParam name="countryproperty" value="Country"/>
    <columnParam name="validator" value="SomeLocalizedString"/>
    <columnParam name="size" value="50"/>
  </column>
</extension>
```

This code adds a column (field) to the `PolicyAddress` entity named `SomeLocalizedString`. For this column, it is necessary to specify the following parameters:

Parameter	Description
countryproperty	Defines a property named <code>Country</code> that exists on the surrounding entity (<code>PolicyAddress</code>) to return the country to use. The value of <code>Country</code> determines the input mask to use.
validator	Defines the field validator to use with this string. The name of the validator is <code>SomeLocalizedString</code> .
size	Defines the width (size) of the new database column.

Step 2: Add the Input Field to the PolicyCenter Interface

The PolicyCenter **Policy Info** page uses the `AccountInfoInputSet` PCF to display information on the primary named insured, including the policy address. This example modifies this PCF to add a new input field named **Some Localized String**. This is merely a generic input field to illustrate the principals involved. In actual practice, you would add a field that had a more meaningful function and label.

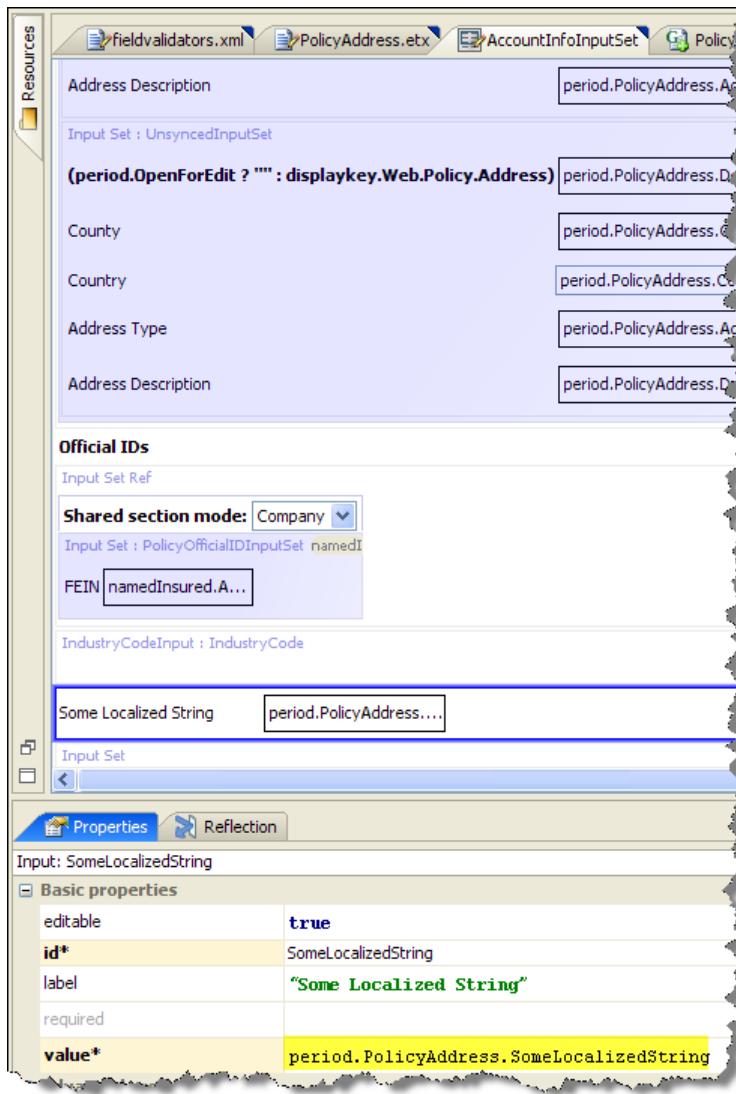
To add the Some Localized String Field

1. Open (in Studio) PCF file `AccountInfoInputSet`.
2. Expand the **Toolbox** and find the **Input** widget. Select this widget and drag it directly underneath the `IndustryCodeInput` widget.
3. Select your **Input** widget to open the **Properties** pane at the bottom of the screen. Enter the following:
 - **id** - `SomeLocalizedString`
 - **label** - "Some Localized String"
 - **value** - `period.PolicyAddress.SomeLocalizedString`

The `value` property indicates that the input field is to use the `PolicyAddress.SomeLocalizedString` column definition. You added this column to `PolicyAddress` by extension in step 1 of this example.

```
column desc="Some Localized String" name="SomeLocalizedString" type="localizedstring">
  <columnParam name="countryproperty" value="Country"/>
  <columnParam name="validator" value="SomeLocalizedString"/>
  <columnParam name="size" value="50"/>
</column>
```

Thus, the input field uses the SomeLocalizedString validator, triggered by the value of the PolicyAddress.Country property. The following graphic illustrates this.



Step 3: Add the Validator to the Default Field Validator File

Next, you need to add the default validator definition to the default validator file.

To add the column validator to the validator files

1. Add the LocalizedString validator to the default `fieldvalidators.xml` file by adding the following line to that file:

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

The LocalizedString data type has an input mask and a regular expression against which it validates. As PolicyCenter retrieves a field validation object for the `PolicyAddress.SomeLocalizedString` column, it looks at this validator.

Step 4: Create Country-Specific Validators and Error Messages

The previous step simply defines a default validator for the **Some Localized String** input field. However, you want the validation input mask to change depending on the country of the policy address. For example, within ClaimCenter:

- If you set the country in an address to the United States, then you want the input mask to be three digits.
- If you set the country to Great Britain, then you want the input mask to be six digits.

You need to add a **SomeLocalizedString** validator to the following files:

File	Location	Description
fieldvalidators_US.xml	fieldvalidatorscountries	Validator to use if <code>PolicyAddress.Country</code> is set to the United States.
fieldvalidators_GB.xml	fieldvalidatorscountries	Validator to use if <code>PolicyAddress.Country</code> is set to Great Britain.

To add the column validator to the validator files

1. Create country-specific versions of the field validators file and place them in the **fieldvalidatorscountries** folder in **Resources → Data Model Extensions**.

For Great Britain and the United States, create the following files and put them in **fieldvalidatorscountries** folder:

- **fieldvalidators_GB.xml**
- **fieldvalidators_US.xml**

2. Place a country-specific version of the **SomeLocalizedString** validator in each of these files.

- For the United States, use the following:

```
<ValidatorDef description="Validator.SomeLocalizedString.US"
    input-mask="###"
    name="SomeLocalizedString"
    value="([0-9]{3})"/>
```

- For Great Britain, use the following:

```
<ValidatorDef description="Validator.SomeLocalizedString.GB"
    input-mask="#####"
    name="SomeLocalizedString"
    value="([0-9]{6})"/>
```

The input mask changes for each country, as does the regular expression. Each country-specific version of the field validators file lives in **fieldvalidatorscountries** and contains only those data type validators specific to that country.

Every time that PolicyCenter renders this field, it checks its country location:

- If the location is in Great Britain, PolicyCenter checks to see if a specific field validator exists in the GB-specific file.
- If the location is in the United States, PolicyCenter checks to see if a specific field validator exists in the US-specific file.
- If a country-specific validator does not exist, then PolicyCenter checks the default field validator file.

Creating Country-Specific Error Messages

If the user enters incorrect input in the **Some Localized String** field, you want the validation error messages to be specific to the country as well. Thus, you need to create country-specific display keys for these error messages and tie them to the validator. To invoke country-specific error messages, you need to create country-specific display keys.

To create country-specific validation error messages

1. Open the **Display Keys** editor and expand the **Validator** node.

2. Right-click and select **Add**. This action opens the **Create Display Key** dialog.
3. Create the following display keys:

Display key name	Value
Validator.SomeLocalizedString.GB	{0} must be six digits (Great Britain).
Validator.SomeLocalizedString.US	{0} must be three digits (United States).

This links each country-specific display key to the country-specific validator definition.

Step 5: Create the Validator Trigger Code

For the example to work correctly, you need to be able to retrieve the `entity.Country` property on the entity linked to the input field. You do this through an entity enhancement property getter method. This is true, in general, for localized entity field validation to work correctly.

However, in the base PolicyCenter configuration, a `PolicyAddressEnhancement` exists already and contains the necessary code to obtain the `PolicyAddress.Country` value. It looks similar to the following:

```
package gw.policyaddress

enhancement PolicyAddressEnhancement : entity.PolicyAddress {
    ...
    property_get Country() : Country {
        if (this.SyncedToAccount) {
            return this.Address.Country
        } else {
            return this.getFieldValue("CountryInternal") as Country
        }
    }
    ...
}
```

Extension file `PolicyAddress.etx` references this property. The property returns the correct country code.

IMPORTANT If a getter method for `Country` property does not exist on the entity linked to your validation field, then you **must** create one. If a Gosu enhancement does not exist for that entity already, then you must create one and populate it with the correct getter method.

Testing the Field Validation Example

After following the example steps, you need to test your configuration to see if it works correctly.

To test the example configuration

1. Restart PolicyCenter Studio, if you have not already done so.
2. Start the PolicyCenter application server. Because of the database changes created in the example, the server needs to upgrade the database. Verify that there are no server errors. If there are, you need to correct the issues (typically mis-typing) before continuing.
3. Log into PolicyCenter and start a new submission using any of the base configuration sample data.
4. Navigate to the **Policy Info** page. If you performed the steps in the example correctly, you see the **Some Localized String** field, in the **Official IDs** group. For an example of how this looks, see the graphics in “Example Set Up” on page 509.
5. Verify that the **Country** field for the policy address is set to either the United States (*United States of America*) or Great Britain (*United Kingdom of Great Britain and N. Ireland*). If it is not, then you need to change the policy address.

- If you set the country to Great Britain, then you see six dots (for six digits) in the **Some Localized Field** input field.
 - If you set the country to the United States, then you see three dots (for three digits) in the **Some Localized Field** input field.
6. Enter one or two digits in the **Some Localized Field** input field and click **Next**. If you have completed the example correctly, you see a localized error message at the top of the screen. The message tells you the number of digits you must enter in this field and varies by which country exists on the policy address.

Localizing Templates

This topic discusses the implications of application localization as it applies to documents, emails, and notes.

This topic includes:

- “Understanding Templates: A Review” on page 517
- “Creating Localized Documents, Emails, and Notes” on page 518
- “Document Localization Support” on page 522

Understanding Templates: A Review

In the base configuration, Guidewire provides a number of template-related definition files for notes, emails, and documents. You can find these files in Studio, in their respective directories, at the following locations:

- Other Resources → **doctemplates**
- Other Resources → **emailtemplates**
- Other Resources → **notetemplates**

Each directory contains, for that resource type, files that ClaimCenter uses to define the document, email, or note. ClaimCenter uses two files (with identical names, but with different file extensions) to define each one.

File extension	Description	Example
.doc	Template file. This file contains the actual content of the document, email, or note.	EmailSent.gosu.htm
.htm		
.pdf		
.xml		
.xls		

File extension	Description	Example
.descriptor	<p>Template descriptor file. This file contains the template metadata, for example:</p> <ul style="list-style-type: none">• Name• ID• MIME type <p>This file can also contain symbol definitions (context objects) that ClaimCenter substitutes into the template content file in creating the final document.</p>	EmailSent.gosu.htm.descriptor

See Also

For general information on templates, how to create them, and how to use them, see:

- “Gosu Templates” on page 291 in the *Gosu Reference Guide*
- “Data Extraction Integration” on page 279 in the *Integration Guide*

Creating Localized Documents, Emails, and Notes

The process of creating localized versions of document, email, and note templates is straight-forward and, for the most part, relatively simple. It mainly involves creating locale-specific folders in the correct location and populating each folder with translated versions of the required document, email, or note templates and descriptor files.

The following steps describe the process.

- Step 1: Create Locale-Specific Folders
- Step 2: Localize Template Descriptor Files
- Step 3: Localize Template Files
- Step 4: Localize Documents, Emails, and Notes within ClaimCenter

Within Guidewire ClaimCenter the default locale for a document, note, or email template is the configured default locale for the application.

Note: Remember that any time you add a file to a Studio-managed file folder, you need to stop and restart Studio so that it recognizes the change.

IMPORTANT Guidewire does not provide the ability to localize Velocity templates.

Step 1: Create Locale-Specific Folders

Guidewire ClaimCenter uses the following directory structure to store the application document, email, and note templates.

*ClaimCenter/modules/cc/config/resources/doctemplates
ClaimCenter/modules/cc/config/resources/emailtemplates
ClaimCenter/modules/cc/config/resources/notetemplates*

As with all other application configuration files, ClaimCenter stores the base configuration version of these files in a *read-only* application directory. To create your own, localized versions of the template files, do the following:

1. Manually copy the application template folder (*doctemplates*, for example) to your configuration directory:
ClaimCenter/modules/configuration/config/resources/

2. Create your locale-specific folders in the configuration directory. For example, if you create a French locale and want to use French-language document, email, and note templates, then you need to duplicate the following directory structure:

```
ClaimCenter/modules/configuration/config/resources/doctemplates/fr_FR/  
ClaimCenter/modules/configuration/config/resources/emailtemplates/fr_FR/  
ClaimCenter/modules/configuration/config/resources/notetemplates/fr_FR/
```

After you complete this task—if you stop and restart Studio—you see your locale-specific folders in the Resources tree, along with all of the non-localized templates. For example:

- Other Resources → doctemplates → fr_FR
- Other Resources → emailtemplates → fr_FR
- Other Resources → notetemplates → fr_FR

IMPORTANT If you create a locale-specific template folder in your configuration directory, then you must copy the entire application template folder and all its contents into the configuration directory as well. If you do not, you cannot see all of the templates in Studio.

See Also

- For a discussion of the language and country codes to use in creating a locale-specific folder, see “Adding a New Locale” on page 468.

Step 2: Localize Template Descriptor Files

After you copy the base configuration templates files to a template locale folder, you then need to create localized versions of the template descriptor files. The manner in which you create localized document template descriptor files is slightly different from the process for creating localized email and note template descriptor files. See the following topics for details:

- Localizing Document Descriptor Files
- Localizing Email and Note Descriptor Files

It is important to understand that localizing template descriptor files serves a different purpose than that of localizing (translating) template content files. The two are different and separate. For example:

- Localizing the subject context object in an email template descriptor file enables a ClaimCenter *user* to see the subject line of that email template in the localized language within ClaimCenter.
- Localizing the content of an email template enables the *recipient* of that email to see its contents in the localized language.

Localizing Document Descriptor Files

The document template descriptor files are XML-based files that conform to the specification defined in file `document-template.xsd`. You can view this file in Studio in the following location:

Other Resources → doctemplates

The descriptor file defines *context objects* (among other items), which are values that ClaimCenter inserts into the document template to replace defined symbols. Thus, ClaimCenter replaces `<%=Subject%>` in the document template with the value defined for this symbol in the descriptor file.

For example, in the base configuration, ClaimCenter contains the following XML definition for the `EmailSent` template descriptor associated with the `EmailSent` document template. Notice that it contains a context object for the subject symbol. You can define as many context objects (and associated symbols) as you want.

```
<?xml version="1.0" encoding="UTF-8"?>  
<DocumentTemplateDescriptor id="EmailSent.gosu.htm" name="Gosu Sample Email Sent Record"  
... keywords="CA, email">  
  
<DescriptorLocalization locale="someLocale" name="localizedName"  
description="localizedDescription" />
```

```

...
<ContextObject name="Subject" type="string">
  <DefaultObjectValue>"localizedSubject"</DefaultObjectValue>
  <ContextObjectLocalization locale="someLocale" display-name="localizedName" />
...
</ContextObject>
...
</DocumentTemplateDescriptor>

```

Localizing a template descriptor file requires that you localize a number of items in the file. The following list describes some of the main items that you want to localize in a descriptor file:

Element	Attribute	Description
<DocumentTemplateDescriptor>	• keywords	Localize the key words associated with this template to facilitate the search for this template within the ClaimCenter search screen.
<DescriptorLocalization>	• locale • name • description	Subelement of <DocumentTemplateDescriptor> - Enter a valid GWLocale value for locale. (See "Step 1: Add the Locale to the Localization File" on page 468 for what constitutes a valid GWLocale.) You can also localize the name and description of this template as it appears within Guidewire ClaimCenter.
<ContextObjectLocalization>	• locale • display-name	Subelement of <ContextObject> - Enter a valid GWLocale value for locale. (See "Step 1: Add the Locale to the Localization File" on page 468 for what constitutes a valid GWLocale.) You can also localize the name of this template as it appears within Guidewire ClaimCenter.

To localize a document template descriptor file, add the appropriate <DescriptorLocalization> and <ContextObjectLocalization> subelements to the file.

IMPORTANT There is only *one* copy of a document template descriptor file. Do **not** create additional copies in locale folders. Instead, add localization elements to the descriptor files in the `doctemplates` folder.

Localizing Email and Note Descriptor Files

The process of localizing the email and note descriptor files is relatively simple. To localize these files, you need only localize the following attributes and place a copy of the file in the correct locale folder.

Element	Attribute	Description
<emailtemplate-descriptor> <notetemplate-descriptor>	• keywords • subject	Localize the key words associated with this template to facilitate the search for this template within the ClaimCenter search screen. Also, localize the subject of this template to show that localized value in ClaimCenter.

To localize an email or note template descriptor file for French (France), first copy to the descriptor file to a `fr_FR` folder. Then, simply localize any key words that you want and the subject tag for the template.

Step 3: Localize Template Files

After creating the locale-specific folder within your configuration directory, you then need to populate the folders with the translated versions of any template content that you intend to use. Unless you want to create a new template, the simplest procedure is to do the following:

- Copy a base language template.

- Translate it into the language of your choice.
- Deploy the translated file to the proper locale-specific configuration folder.

Note: Any time that you add a file to a Studio-managed file folder, you need to stop and restart Studio so that it recognizes the change.

Step 4: Localize Documents, Emails, and Notes within ClaimCenter

After you create localized versions of your templates, you can then use these templates within ClaimCenter to create a language-specific version of a document, an email, or a note.

Note: Within ClaimCenter, you can select the un-localized templates that are in the default directory.

ClaimCenter displays these templates with no language specified. If you select one, however, ClaimCenter makes the **Language** field in the **New Document** worksheet editable.

To create a localized document

1. Within the ClaimCenter **Claim** tab, select **Create New Document...** **Create from a template** from the **Actions** menu. This action opens the **New Document** worksheet at the bottom of the screen.
2. In the **New Document** worksheet, click the search icon next to the **Select Template** field.

IMPORTANT The base configuration *Sample Acrobat* document (*SampleAcrobat.pdf*) uses Helvetica font. If you intend to create a document that uses Unicode characters (for example, one that uses an East Asian language), then the document template must support a Unicode font. Otherwise, the document does not display Unicode characters correctly.

3. In the search screen that opens, set the **Language** field to your desired language and set the other search fields as necessary. If a document template for your required language exists, ClaimCenter displays it in **Search Results**.
4. Click **Select**. ClaimCenter returns to the **New Document** worksheet with the selected localized template.
5. Complete the rest of the worksheet fields as necessary. You can enter text in your desired language in the appropriate fields to further localize the document.

To create a localized email

1. Within ClaimCenter, select **New... Email** from the **(Claim tab)** **Actions** menu. This action opens the **Email** worksheet at the bottom of the screen.
2. In the **Email** worksheet, you can either enter text in your desired language, or click **Use Template** to open the template selection worksheet.
3. In the search screen that opens, set the **Language** field to your desired language and set the other search fields as necessary. If an email template for your required language exists, ClaimCenter displays it in **Search Results**.
4. Click **Select**. ClaimCenter returns to the **New Email** worksheet with the selected localized template.
5. Complete the rest of the worksheet fields as necessary.

To create a localized note

1. Within ClaimCenter, select **New... Note** from the **(Claim tab)** **Actions** menu. This action opens the **Note** worksheet at the bottom of the screen.
2. In the **Note** worksheet, you can either enter text in your desired language, or click **Use Template** to open the template selection worksheet.
3. In the search screen that opens, set the **Language** field to your desired language and set the other search fields as necessary. If a template for your required language exists, ClaimCenter displays it in **Search Results**.

4. Click **Select**. ClaimCenter returns to the **New Note** worksheet with the selected localized template.
5. Complete the rest of the worksheet fields as necessary. You can enter text in your desired language in appropriate fields to further localize the document.

You can optionally link a localized document to your localized note as well. To do so, click **Link Document**.

Document Localization Support

ClaimCenter provides a number of useful methods for working with document localization in the following APIs.

- **IDocumentTemplateDescriptor**
- **IDocumentTemplateSource**
- **IDocumentTemplateSerializer**

See Also

- “Document Creation” on page 151 in the *Rules Guide* for general information on APIs used in document creation.
- “Document Localization Support” on page 522 for information on specific APIs related to document localization.
- “Document Management” on page 249 in the *Integration Guide* for integration-related issues in document creation.

IDocumentTemplateDescriptor

The **IDocumentTemplateDescriptor** API provides the following getter methods to use with document locale.

Method	Return type	Returns
<code>getLocale()</code>	<code>String</code>	Locale in which to create the document from this template descriptor. A return value of <code>null</code> indicates an unknown language. In most cases, the method returns the default language for the application.
<code>getName(locale)</code>	<code>String</code>	Localized name for a given <code>locale</code> . If a translation does not exist, the method returns <code>null</code> .

You can also use the following property on **IDocumentTemplateDescriptor** to retrieve the locale:

```
gw.plugin.document.IDocumentTemplateDescriptor.locale
```

IDocumentTemplateSource

The **IDocumentTemplateSource** API provides the following useful getter methods.

Method	Return type	Returns
<code>getDocumentTemplate(date, valuesToMatch, maxResults)</code>	<code>IDocumentTemplateDescriptor[]</code>	Array of <code>IDocumentTemplateDescriptor</code> objects. If you need to perform a search based on the locale of a document template, you can specify this in the <code>valuesToMatch</code> argument.
<code>getDocumentTemplate(templateId, locale)</code>	<code>IDocumentTemplateDescriptor</code>	Document template instance corresponding to the specified template ID and locale.

Method	Return type	Returns
<code>getTemplateAsStream(templateId, locale)</code>	<code>InputStream</code>	One of the following: <ul style="list-style-type: none">• Returns an <code>InputStream</code> for reading the specified template• Returns <code>null</code> if ClaimCenter cannot find the template• Returns <code>null</code> if the caller does not have adequate privileges to retrieve the template

IDocumentTemplateSerializer

The `IDocumentTemplateSource` API provides the following method that you can use to retrieve a localized version of a template.

Method	Return type	Returns
<code>localize(locale, descriptor)</code>	<code>IDocumentTemplateDescriptor</code>	Localized instance of the specified template

Localized Search and Sort

This topic describes how to configure a Guidewire ClaimCenter implementation to perform localized search and sort operations. Guidewire provides support for locale-appropriate searching and sorting for a single locale for a Guidewire ClaimCenter installation.

This topic includes:

- “Searching and Sorting Character Data” on page 525
- “How Guidewire Stores Data” on page 526
- “Working with the Default Application Locale” on page 526
- “Configuring a Locale” on page 526
- “Database Searching” on page 527
- “Data Sorting” on page 530

Searching and Sorting Character Data

There are several different ways to search and sort character data:

- One way is to treat the character data as binary code points and compare and sort the data numerically.
- Another way is to treat the character data linguistically. This approach applies specific collation rules (rules used to order words within a list) that reflect the commonly accepted practices and expectations for a particular region.

Linguistic search applies a specific collation *strength* to the character data. Collation strength refers to how selective (or restrictive) the search and sort code treats the collation process. For example, collation strength controls whether the search and sort code respects or ignores differences in case and accent on a character (such as the leading character on a word). In the Japanese language, collation strength also controls whether the search and sort code respects or ignores the differences between *katakana* and *hiragana*, and full- and half-width *katakana* characters differences.

Guidewire ClaimCenter uses the value of configuration parameter `DefaultApplicationLocale` and the `localization.xml` and `collations.xml` configuration files to implement localized search and sort functionality.

How Guidewire Stores Data

Guidewire ClaimCenter stores character data in several different ways:

- Database storage
- In-memory storage

ClaimCenter handles searching and sorting of character data differently depending on the storage medium on which the data resides.

Database Data Storage

Guidewire ClaimCenter writes most application data directly to the database. This action stores the data in a more permanent form, on a physical disk storage system. Each discrete piece of data is an entry in a table column, with each table being organized by rows. During a comparison and sort of data in the database, the database management system (DBMS) performs the operations and applies rules that control these operations.

In-memory Data Storage

Guidewire ClaimCenter writes some application data to volatile memory devices, such as the local machine RAM memory. Guidewire typically uses this kind of memory storage for the display of certain kinds of data in the ClaimCenter interface. For example, ClaimCenter uses in-memory storage for the results of non-query based list views, drop-downs, and other similar items. During a comparison and sort of data in memory, programming code (as implemented by Guidewire) controls the operations.

Working with the Default Application Locale

Your ClaimCenter installation provides support for locale-appropriate searching and sorting for a *single locale only*. ClaimCenter reads this locale from the `DefaultApplicationLocale` configuration parameter, which you set in `config.xml`. The default is United States English (`en_US`).

You set this value one-time only, before you start the application server for the first time. ClaimCenter stores this value in the database and checks the value at server start up. If an application server value does not match a database value, then ClaimCenter throws an error and refuses to start.

WARNING You set configuration parameter `defaultApplicationLocale` one-time only. You must set the value of this configuration parameter **before** you start the application server for the first time. Any attempt to modify this parameter thereafter can invalidate your Guidewire installation.

See Also

- “`DefaultApplicationLocale`” on page 53

Configuring a Locale

You define and manage locales in file `localization.xml`, which you access in Studio in the **Other Resources** folder. This file contains one or more `<GWLocale>` elements, each of which defines the application behavior for a particular country and language. Defining a locale in `localization.xml` enables its use and defines its date, time, number, and currency formats.

The `<GWLocale>` element contains several *optional* subelements, which you can use to configure the behavior of searching and sorting operations in an Oracle database. *It is important to understand that these configuration elements control the behavior of search and sort algorithms only if you are using an Oracle database.*

IMPORTANT For SQL Server, the collation set on the database itself controls the behavior of the search and sort algorithms in the SQL Server database.

See Also

- “Adding a New Locale” on page 468 for a general discussion of `GWLocale`.

Database Searching

How Guidewire ClaimCenter handles searching of data depends on the database involved. See the following:

- Searching and the Oracle Database
- Searching and the SQL Server Database

Searching and the Oracle Database

To be eligible for inclusion in the database search algorithm, the `supportsLinguisticSearch` attribute on that column must be set to `true`. Setting this column attribute to `true` marks that column as searchable. This is true independent of which DBMS you are using.

Note: See “`<column>`” on page 212 for more information on the `<column>` element and the attributes that you can set on it.

File `localization.xml`

For Oracle, Guidewire provides the ability to configure locale-appropriate linguistic search capabilities through the `<LinguisticSearchCollation>` element on `<GWLocale>` in `localization.xml`. You use the `strength` attribute on this optional subelement to configure and control specialized search behavior.

IMPORTANT Any change to the `<LinguisticSearchCollation>` element in `localization.xml` requires a database upgrade. If you make a change to this element, then you must restart the application server to force a database upgrade.

The meaning of the `strength` attribute depends on the specific locale. But, in general,

- A `strength` of `primary` considers only character weights. This setting instructs the search algorithms to consider just the base, or primary letter, and to ignore other attributes such as case or accents. Thus, the collation rules consider the characters é and E to have the same weight.
- A `strength` of `secondary` considers character weight and accent differences, but, not case differences. Thus, the collation rules consider the characters e and é to be different and thus the rules treat them differently. This is the default value.

To summarize, the `strength` attribute can take the following values, with the default being `secondary`.

Strength	Searches are
primary	<ul style="list-style-type: none">• accent-insensitive• case-insensitive
secondary	<ul style="list-style-type: none">• accent-sensitive• case-insensitive

The following `localization.xml` file is an example of this file with suggested settings.

```
<?xml version="1.0" encoding="UTF-8"?>
<Localization xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
               xsi:noNamespaceSchemaLocation="../../../../../../platform/pl/xsd/localization.xsd">
    <!-- United States English; the default if nothing else is modified -->
    <GWLocale name="English (US)" code="en_US" typecode="en_US">
        <DateFormat short="MM/dd/yyyy"
                     medium="MMM d, yyyy"
                     long="E, MMM d, yyyy" />
        <TimeFormat short="hh:mm aa"
                     medium="hh:mm aa"
                     long="hh:mm aa"/>
        <NumberFormat decimalSymbol="."
                     thousandsSymbol="," />
        <CurrencyFormat positivePattern="#"
                         negativePattern="($#)"
                         zeroValue="-" />
        <LinguisticSearchCollation strength="primary"/>
        <SortCollation strength="tertiary"/>
    </GWLocale>
</Localization>
```

File collations.xml

Guidewire provides specialized search rules for use with the Oracle database through the use of a configurable Java class that it exposes as a `CDATA` element in the `collations.xml` file. You access `collations.xml` in Studio in the **Other Resources** folder.

Within this file, search for the following:

```
<Database type="ORACLE">
    ...
    <DBJavaClass> <![CDATA[...]]></DBJavaClass>
    ...
</Database>
```

Guidewire ClaimCenter uses this Java code as the source code for a Java class in Oracle. Oracle then uses this Java class to implement the calculation of the denormalization fields in Oracle for primary-strength searching and for Japanese- and German-language searching.

In the base configuration, this Java class defines:

- General rules for primary-strength searching in the Oracle database
- Specialized rules for searching in the Japanese language
- Specialized rules for searching in the German language

General Search Rules for the Oracle Database

In the base configuration, ClaimCenter uses the following general rules as it performs a search on an Oracle database:

- All searches are case insensitive. This is regardless of the value of the `strength` attribute on `<LinguisticSearchCollation>`. ClaimCenter regards the characters e and E as the same.
- All searches take punctuation into account. ClaimCenter does **not** regard O'Reilly as equal to OReilly.
- All searches in which the `strength` attribute on `<LinguisticSearchCollation>` is set to `primary` ignore accent—or diacritic—marks. ClaimCenter regards the characters e and è as the same in this type of search.
- All searches in which the `strength` attribute on `<LinguisticSearchCollation>` is set to `secondary` take into account any accent marks. ClaimCenter does **not** regard the characters e and è as the same in this type of search.

IMPORTANT ClaimCenter searches only on database columns on which you set the `supportsLinguisticSearch` attribute to `true` (`supportsLinguisticSearch="true"`)

General Search Rules for the Japanese Language

In the base configuration, Guidewire provides specialized search algorithms specifically for the Japanese language. Guidewire sets these rules through the use of a configurable Java class that it exposes as a CDATA element in the `collations.xml` file. You can access this file in Studio in the **Other Resources** folder. After you open the file, search for the following:

```
<Database type="ORACLE">
  ...
  <DBJavaClass> <! [CDATA[...]]></DBJavaClass>
  ...
</Database>
```

This Java class provides the following behavior for searching in a Japanese-language Oracle database:

Search case	Rule
Half-width/Full-width	All searches in Japanese ignore the difference between half-width and full-width Japanese characters.
Small/Large characters	All searches in Japanese in which the <code>strength</code> attribute on <code><LinguisticSearchCollation></code> is set to <code>primary</code> (meaning accent-insensitive) ignore Japanese small/large letter differences in Katakana or Hiragana. Searches in which this attribute is set to <code>secondary</code> take small/large letter differences into account.
Katakana and Hiragana	All searches in Japanese ignore the difference between <i>katakana</i> and <i>hiragana</i> characters. This is also known as <i>kana</i> -insensitive searching.
Long dash (—)	All searches in Japanese ignore the long dash character.
Sound marks (` and °)	All searches in Japanese in which the <code>strength</code> attribute on <code><LinguisticSearchCollation></code> is set to <code>primary</code> ignore sound marks. Searches in which this attribute is set to <code>secondary</code> take sound marks into account.

IMPORTANT If you modify the contents of `collations.xml` or the embedded Java class, ClaimCenter forces a database upgrade the next time the application server starts.

General Search Rules for the German Language

In the base configuration, Guidewire provides specialized search algorithms specifically for the German language. Guidewire sets these rules through the use of a configurable Java class that it exposes as a CDATA element in the `collations.xml`. This is the same Java class that the discussion on rules for the Japanese language covered.

This Java class provides the following behavior for searching in a German-language Oracle database:

Search case	Rule
Vowels with umlauts	All searches in German compare as equal a vowel with an umlaut or the same vowel without the umlaut but followed by the letter e. Thus, all searches in German explicitly treat the following as the same value: <ul style="list-style-type: none">• ä and ae• ö and oe• ü and ue
German letter Eszett	All searches in German treat the Eszett character ß (also known as Sharp-S) the same as the characters ss.

Searching and the SQL Server Database

For SQL Server, ClaimCenter always performs a *case-insensitive* database search. The search algorithm obeys the rules for the Windows collation set on the database only.

IMPORTANT Guidewire recommends that you determine your choice of database collation **before** you start the ClaimCenter application server for the first time. In other words, set the collation on the database before you populate it with any data.

Data Sorting

How Guidewire ClaimCenter handles the sorting of character data depends on the database involved. See the following:

- Sorting and the Oracle Database
- Sorting and the SQL Server Database

Note: ClaimCenter uses in-memory sorting in the application interface for non-query based list views, drop-down lists, and similar items. To perform in-memory sorting, application code uses a `Collator` object obtained from the locale and modified with the configured `strength` value of the `<SortCollation>` element from file `localizaton.xml`.

Sorting and the Oracle Database

For Oracle, Guidewire treats the ordering of data as consistently as possible between database sorting and in-memory sorting. For optional use, `<SortCollation>` on `<GWLocale>` in `localization.xml` controls specialized sorting behavior. This element contains a single `strength` attribute that determines how ClaimCenter handles accents and case during the sorting of character data for the following:

- Sorting of in-memory data
- Sorting of data in the Oracle database

IMPORTANT For Oracle, the `<SortCollation>` element controls sort behavior for both in-memory sorting and database sorting. In contrast, for SQL Server, this element controls only the sorting of data in memory, as database collation—which you set on the database itself—controls database sorting.

The `<SortCollation>` Subelement

The `<GWLocale>` element includes an optional subelement—`<SortCollation>`—that you use to set collation strength in sorting algorithms through its `strength` attribute. This attribute can take the following values, with the default being `secondary`.

- `primary`
- `secondary`
- `tertiary`

The meaning of the `strength` attribute depends on the specific locale. But, in general,

- A `strength` of `primary` considers only character weights. This setting instructs the search and sort algorithms to consider just the base, or primary letter, and to ignore other attributes such as case or accents. Thus, the collation rules consider the characters `e` and `E` to have the same weight.
- A `strength` of `secondary` considers character weight and accent differences. This is the default. Thus, the collation rules consider the characters `e` and `è` to be different and orders them differently.
- A `strength` of `tertiary` considers character weight, accent differences, and case. Thus, the collation rules consider the characters `e` and `è` and `E` to be different and orders them differently.

The following list describes these differences.

Strength	Case-sensitive	Accent-sensitive
primary	No	No
secondary	No	Yes
tertiary	Yes	Yes

IMPORTANT ClaimCenter sorts search page results by the denormalized data column. Thus, ClaimCenter displays search results using the rules and the strength of the search collation and **not** the strength of the sort collation.

File `collations.xml`

Guidewire uses `collations.xml` as a *lookup* file. You access `collations.xml` in Studio in the **Other Resources** folder. Guidewire ClaimCenter uses the following definitions in this file to look up the sort collation name and apply it. ClaimCenter uses:

- the default application locale
- the `strength` value from the `<SortCollation>` element in `localization.xml`
- the database management system (DBMS) type

Primarily, ClaimCenter uses these values to look up the Oracle sort collation, which it then sets as the value for `NLS_SORT` for all Oracle database sessions. This ensures that all Oracle database sorting uses the rules specified by that collation.

For example, suppose that the following are all true:

- The database is Oracle.
- The user locale is German.
- The `strength` value of `SortCollation` (in `localization.xml`) is set to `secondary`.

ClaimCenter then looks at the following for instructions on how to set `NLS_SORT` for Oracle sessions and sets it to `GERMAN_CI`.

```
<Database type="ORACLE">
  ...
    <Collation locale="de" primary="GERMAN_AI" secondary="GERMAN_CI" tertiary="GERMAN"/>
  ...

```

Secondarily, ClaimCenter uses this file to support the ordering of typekeys from the database. In sorting typekeys, Guidewire uses the following criteria:

- ClaimCenter looks first at the priority associated with the type key and orders the typekeys by priority order.
- If there are typekeys with the same priority, ClaimCenter applies the locale collation rules to the typekey names of the same priority.

ClaimCenter applies the collation rules to the typekey columns in database query `ORDER BY` clauses that sort database query results. Thus, file `collations.xml` contains multiple locale collations as ClaimCenter supports storing typekey values in multiple languages in one database. This enables different users with various locale settings to see different translations of a typekey and for ClaimCenter to sort the typekey names correctly.

Sorting and the SQL Server Database

It is important to understand that the optional `<SortCollation>` element on `<GWLocale>` in `localization.xml` has no effect on a SQL Server database. *This value does not control the sorting of data in the SQL Server database.* Instead, database collation—as set in the database itself as you create it—controls the sorting of data in that database.

For the SQL Server database:

- Guidewire **does** require that the SQL Server database collation be set to case-insensitivity (CI). This collation controls both searching and sorting. For the Japanese language, the Microsoft Windows collations default to width- and Kana-insensitivity.
- Guidewire does **not** require a specific setting for accented characters. It is your choice, depending on your business needs, whether you need to set database search collation to accent-insensitive as well.

IMPORTANT Guidewire recommends that you determine your choice of database collation **before** you start the ClaimCenter application server for the first time. In other words, set the collation on the database before you populate it with any data.

Localizing Addresses

Many of the Guidewire ClaimCenter screens display address information. This topic provides information on the following:

- How ClaimCenter supports address localization
- How to set a mode on an address to format the address for a specific country
- How to configure and use PCF address widgets

This topic includes:

- “Address Localization Overview” on page 533
- “The AddressOwner and CCAddressOwner Interfaces” on page 534
- “The CCAddressOwnerId and CountryAddressFields Classes” on page 535
- “PCF Address Configuration” on page 537

Address Localization Overview

Many of the Guidewire ClaimCenter screens display address information. This address information poses a challenge both for maintenance and for localization. To facilitate ease of use in working with address information, ClaimCenter provides single address view that can have a separate mode for each country code.

The address view provides a single view that you can configure to display an address differently depending on the country used in the address. For example, if you want to use a different international address for Germany, you need merely to configure the address view and make it modal by country code.

ClaimCenter builds the address view around the concept of an address owner. The address owner (as the name implies) is the object that owns a particular address. This concept ensures that ClaimCenter saves the address properly. For example:

- If the address is a loss location, then ClaimCenter saves the location address with a specific `Claim` object.
- If the address is a temporary location for loss of use, then ClaimCenter saves the location address with a specific `Exposure` object.

The Address User Interface

The address interface is consistent across ClaimCenter:

- In *view mode*, ClaimCenter displays the address on two lines. The description field is available for all addresses. You can configure this (in Studio) from **Entity Names** → **Address**, by modifying the Gosu code in the **Entity Name** definition field.
- In *edit mode*, ClaimCenter displays the address as separate, editable, input fields.

ClaimCenter always displays a range selector (a drop-down list) in the address user interface so that you can select from an appropriate address array. If you cannot edit an address, then ClaimCenter indicates this by marking the address fields gray. The drop-down list can also display a **New...** command that you can use to create a new address.

The *AddressOwner* and *CCAddressOwner* Interfaces

In the base configuration, Guidewire provides the following address-related interface:

`CCAddressOwner`

If an entity type contains an address, then that entity type must extend the `CCAddressOwner` interface. In actual practice, interface `CCAddressOwner` extends interface `AddressOwner` to provide a merged set of properties and methods that you use to manage address-related objects.

The *AddressOwner* Interface

The `AddressOwner` interface creates a helper object that it passes to the `AddressInputSet` object. The helper object provides a way to set and get a single address on the enclosing entity. It also provides methods that you can use to set a field as required or visible, for example. The following list describes the available properties on `AddressOwner`.

Property	Description
<code>Address</code>	Sets (or retrieves) a single address on the enclosing entity. For example, you can use this to set or get the primary address for a <code>Contact</code> . ClaimCenter automatically creates a new <code>Address</code> object if you use a Gosu expression of the form: <code>owner.Address.State = someState</code>
<code>HiddenFields</code>	Set of address fields that ClaimCenter hides (does not show) in the application interface.
<code>RequiredFields</code>	Set of address fields for which the user must supply a value.

The *CCAddressOwner* Interface

The `CCAddressOwner` interface extends the `AddressOwner` interface to add ClaimCenter specific fields. To be useful, you must then create a class that implements that `CCAddressOwner` interface. This class must provide method bodies for all `CCAddressOwner` interface methods.

You pass a `CCAddressOwner` object as the argument to the generic `AddressInputSet` object. The generic `AddressInputSet` calls methods and properties on `CCAddressOwner` to determine the following (for example):

- Which fields to show as visible on the screen
- Which fields require user input on the screen
- How to get and set the `Address` object

The use of this `CCAddressOwner` interface enables you to reuse a single model `AddressInputSet` in different situations. This makes it relatively simple to add new countries to the system.

The following list describe the properties and methods available on `CCAddressOwner`. For more details on how these methods work, consult the comments in the `CCAddressOwner` Gosu class itself. You can access the interface definition file through Guidewire Studio at `gw.address.CCAddressOwner`.

Property or method	Description
Addresses	Array of pre-populated addresses that ClaimCenter shows in a drop-down list from which the user can select. If this method returns null, then ClaimCenter does not show a drop-down selection list.
AddressNameLabel	The label (text string) to use for one of the following: <ul style="list-style-type: none"> • The range input to use in picking an address from a list of addresses • The summary of an address Only one of these can be visible at one time. You only see either the range input or the address summary.
Claim	If non-null, then the address input set also displays additional claim-specific fields: <ul style="list-style-type: none"> • The claim loss location code, <code>Claim.LossLocationCode</code>, a simple string field. • The claim jurisdiction state, <code>Claim.JurisdictionState</code>. These fields are only visible if <code>CCAddressOwner.Claim</code> returns a non-null value and if a field is not in the <code>HiddenFields</code> list.
ConfirmCountryChange	Boolean value that indicates whether to display a confirmation message if the user changes the address country. This value is usually set to true, except on search screens.
DefaultCountry	Sets that default country to use if the current address is null.
getOrCreateNewAddress()	Method that retrieves an existing address or creates a new address for use in the address drop-down list. ClaimCenter calls this method if the user selects New... from the address drop-down list.
InputSetMode	The mode to use for the address input set. This is nearly always the same as the selected country. The only exception is if the address input set is being used as part of a search screen. In that case the default country and selected country can both return null. (It is common to perform a search operation with no country set.)
NonEditableAddresses	Set of addresses that you cannot edit. If an address appears in the Addresses array and also in NonEditableAddresses, it is possible to select it from the drop-down list of addresses. However, the address fields are read-only.
SelectedCountry	Sets (or retrieves) the currently selected country. For example, the following sets the currently selected country to France: <code>adrs.SelectedCountry = "FR"</code>
ShowAddressSummary	Boolean value that sets whether ClaimCenter displays the address as a summary instead of separate input fields.

The CCAddressOwnerId and CountryAddressFields Classes

In the base configuration, Guidewire provides the following classes that are useful in working with addresses:

- `CountryAddressFields`
- `CCAddressOwnerId`

In general, you can configure address fields in the following ways:

- You can make country-specific changes by adding a country-specific `CountryAddressFields` object.
- You can make simple global changes by altering the values of the set of constants in `CCAddressOwnerId`.

You can also add to or alter the particular `CCAddressOwner` object that you pass as a parameter to the `AddressInputSet` object. In this way, you can use the passed-in owner object to override any country-specific or global defaults that you have already set.

Class *CountryAddressFields*

In the base configuration, Guidewire provides a single `CountryAddressFields` class that is general enough for you to use for all countries in a simple configuration. To add a country-specific `CountryAddressObject`, create a new class in package `gw.api.address.countryfields` that extends `CountryAddressFields` and use the following naming convention for the subclass:

```
CountryAddressFields_XX
```

In this case, XX is the country code, which is, for example, US, AU, GB, or some other standard country code.

Your `CountryAddressFields` subclass must have a constructor that takes no arguments, for example:

```
package gw.api.address.countryfields
uses gw.api.address.CountryAddressFields

class CountryAddressFields_GB extends CountryAddressFields {
    construct() { super("GB") }
    ...
}
```

Your subclass can override any of the methods provided by the super class, `CountryAddressFields`. As you only need to instantiate this subclass a single time (after which time, it is shared), do not create a class that contains any mutable state. The class must only return constant values.

The general usage of the `CountryAddressFields` class is to create a Gosu expression similar to the following:

```
CountryAddressFields.forCountry(country).getClaimFileRequiredFields(claim)
```

This code calls `forCountry` to retrieve the `CountryAddressFields` object for a particular country. It then calls a method on that object to retrieve the required or hidden fields for a particular context for that country.

Class *CCAddressOwnerId*

Guidewire provides a `gw.api.address.CCAddressOwnerId` class that you can use to configure address formatting on a country-specific basis. Using this class, it is possible to set—country by country—the fields that are visible on the ClaimCenter screen and the fields that require user input.

The `CCAddressOwnerId` class contains the following set of constants. For more details on the meaning of these properties, consult the comments in the `CCAddressOwnerId` Gosu class itself. You can access the interface definition file through Guidewire Studio at `gw.address.CCAddressOwnerId`.

- ADDRESS_NAME
- ADDRESSLINE_1
- ADDRESSLINE_2
- ADDRESSLINE_3
- ADDRESSTYPE
- CITY
- CITY_STATE_REQUIRED
- COUNTRY
- COUNTY
- DESCRIPTION
- JURISDICTIONSTATE
- LOCATIONCODE
- NO_FIELDS
- NON_ADDRESS_FIELDS
- POSTALCODE
- POSTALCODE_REQUIRED
- SEARCH_HIDDEN_FIELDS
- STATE
- VALIDUNTIL
- VALIDUNTIL_ADDRESSTYPE_HIDDEN
- WC_HIDDEN_FIELDS

PCF Address Configuration

ClaimCenter stores the shared address PCF views in the following location in Studio:

Resources → Page Configuration (PCF) → claim → shared → address

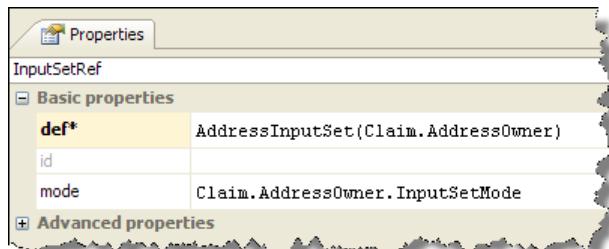
Each AddressInputSet takes the form of AddressInputSet.XX, with the XX suffix being one of the following:

- default—The default address format to use if the country that you select from the ClaimCenter locale drop-down does not exist as a mode.
- XX—The two-letter country code that indicates which address input mode to use in formatting this address.

For example, you use AddressInputSet.US to display addresses formatted for the United States and AddressInputSet.CA for addresses formatted for Canada. The following list illustrates some of the differences in address formatting used in the different countries.

AddressInputSet.US	AddressInputSet.CA
Address 1	Address 1
Address 2	Address 2
City	City
County	
State	Province
ZIP Code	Postal Code

You reference an AddressInputSet by using an InputSetRef widget. For instance, you see the following in the LossDetailsDV.Auto.pcf. The InputSetRef widget on this page displays loss location information.



In this example, the AddressInputSet takes a single parameter and a mode:

Property	Value	Meaning
def	Claim.AddressOwner	The CCAddressOwner object that you pass to the AddressInputSet
mode	Claim.AddressOwner.InputSetMode	The switch that selects the view mode

In the example, the Claim.AddressOwner parameter that you pass to AddressInputSet is a CCAddressOwner object. The mode is the switch to select the correct view mode. If a country that you select from the ClaimCenter drop-down locale does not exist as a mode, then ClaimCenter uses AddressInputSet.default.pcf.

Gosu Configuration for a New Address Relationship

It is possible to create a AddressOwner delegate on an entity if you add an Address foreign key to a new or existing entity.

1. Create a new Gosu class that implements the CCAddressOwner interface. For example, suppose that you create a new class and call it CustomEntityAddressOwner.

- a. Select the gw.entity package (in the Classes folder), right-click, and select New → Class.

- b. Enter `CustomEntityAddressOwner` in the New Class dialog. This action creates a new `CustomEntityAddressOwner` class file.

- c. Add the following uses statements directly underneath the package statement.

```
uses gw.api.address.CCAddressOwner
uses gw.api.address.AddressOwnerFieldId
uses java.util.Set
```

- d. Add the following to the initial class definition line:

```
class CustomEntityAddressOwner implements AddressOwner
```

- e. Place the mouse cursor within the `class` text and hover the mouse near the `implements` word. Studio prompts you to implement the required interface methods by pressing ALT+ENTER.

- f. After pressing ALT+ENTER. Studio automatically inserts the necessary default methods.

```
package gw.entity

uses gw.api.address.AddressOwnerFieldId
uses java.util.Set
uses gw.api.address.CCAddressOwner

class CustomEntityAddressOwner implements CCAddressOwner {

    construct() { }

    override property get Address() : Address {
        return null //## todo: Implement me
    }

    override property set Address(value : Address) {
        //## todo: Implement me
    }

    override property get RequiredFields() : Set<AddressOwnerFieldId> {
        return null //## todo: Implement me
    }

    override property get HiddenFields() : Set<AddressOwnerFieldId> {
        return null //## todo: Implement me
    }

    override property get Addresses() : Address[] {
        return null //## todo: Implement me
    }

    override function getOrCreateNewAddress() : Address {
        return null //## todo: Implement me
    }

    override property get NonEditableAddresses() : Set<Address> {
        return null //## todo: Implement me
    }

    override property get Claim() : Claim {
        return null //## todo: Implement me
    }

    override property get DefaultCountry() : Country {
        return null //## todo: Implement me
    }

    override property get SelectedCountry() : Country {
        return null //## todo: Implement me
    }

    override property set SelectedCountry(value : Country) {
        //## todo: Implement me
    }

    override property get InputSetMode() : Country {
        return null //## todo: Implement me
    }

    override property get ShowAddressSummary() : boolean {
        return null //## todo: Implement me
    }
}
```

```
override property get ConfirmCountryChange() : boolean {  
    return null //## todo: Implement me  
}  
  
override property get AddressNameLabel() : String {  
    return null //## todo: Implement me  
}  
}
```

See `VehicleIncidentAddressOwner.gs` and `ClaimRelatedAddressOwner.gs` for implementation examples.

2. Create a new Gosu enhancement to enhance the new or existing entity with `get` and `set` methods. For example, see `GWTripAccommodationAddressOwnerEnhancement.gsx` for sample implementation details.

```
package gw.entity  
  
uses gw.api.address.TripAccommodationAddressOwner  
uses gw.api.address.CCAddressOwner  
  
enhancement GWTripAccommodationAddressOwnerEnhancement : TripAccommodation {  
    property get AddressOwner() : CCAddressOwner { return new TripAccommodationAddressOwner(this) }  
}
```


Working with the Japanese Imperial Calendar

This topic discusses the Japanese Imperial Calendar and how to configure Guidewire ClaimCenter to access and display Japanese Imperial Calendar dates correctly.

This topic includes:

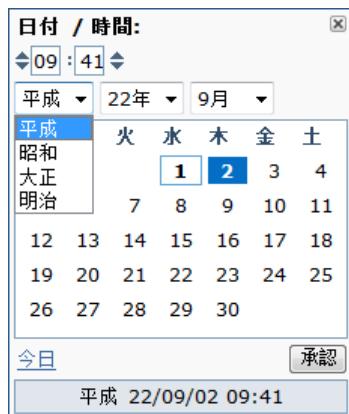
- “The Japanese Imperial Calendar Date Widget” on page 541
- “Configuring Japanese Dates” on page 542
- “Setting the Japanese Imperial Calendar as the Default for a Locale” on page 543
- “Working with the Japanese Imperial Date Data Type” on page 543
- “Setting a Field to Always Display the Japanese Imperial Calendar” on page 544
- “Setting a Field to Conditionally Display the Japanese Imperial Calendar” on page 545
- “Sample JIC Presentation Handler” on page 546

The Japanese Imperial Calendar Date Widget

The ClaimCenter interface supports the use of a Japanese Imperial calendar date widget. You must enable this feature through configuration. After you enable it, you can specify the default calendar type for the following:

- An entity property
- A Java or Gosu property

The following graphic illustrates the *Date picker* for the Japanese Imperial calendar:



The date widget only displays the four most recent Imperial eras. They are:

- *Heisei*
- *Showa*
- *Taisho*
- *Meiji*

IMPORTANT Guidewire does not support the use of an input mask for Japanese Imperial calendar input. Enter date input through the use of the date picker. If you use a copy-and-paste method to enter a date, then you must paste an exact matching string into the date field.

Configuring Japanese Dates

IMPORTANT Guidewire uses the *International Components for Unicode* (ICU) open source libraries to support the Japanese Imperial calendar. The ICU libraries hard-code the historical Imperial data. Therefore, any change to the calendar—for example, the start of a new era—requires an upgrade to the calendar ICU libraries. If this is an issue, contact Guidewire support for details of how to upgrade the library.

Note: For information on how to set up a Japanese locale, see “Adding a New Locale” on page 468.

You configure how ClaimCenter displays Japanese Imperial calendar dates by specifying the `<JapaneseImperialDateFormat>` element in `localization.xml`. The `<JapaneseImperialDateFormat>` element is similar to the existing Gregorian `<DateFormat>` element. You need to define the following attributes:

Date format	Description
long medium	ClaimCenter only uses the long and medium date formats to display Japanese dates. You cannot use these fields to format user input data.
short	ClaimCenter uses the short date format to format user date input. Any date input format pattern that you choose must be compatible with the fixed width of the input mask. Guidewire recommends that you use <code>short="yy/MM/dd"</code> .
yearSymbol	ClaimCenter uses the Japanese yearSymbol as a signal to render the Japanese Imperial calendar date picker.

The following graphic illustrates a Japanese <GWLore> definition along with the associated <JapaneseImperialDateFormat> element.

```
<!-- Japanese (Japan) -->
<GWLore name="Japanese (Japan)" code="ja_JP" typecode="ja_JP" defaultCalendar="JapaneseImperial">
    <DateFormat short="yyyy-MM-dd"
        medium="yyyy-MM-dd"
        long="EEEE, yyyy年M月d日"/>
    <TimeFormat short="hh:mm aa"
        medium="HH:mm:ss"
        long="aa h時mm分ss秒"/>
    <NumberFormat decimalSymbol="."
        thousandSymbol=","/>
    <CurrencyFormat positivePattern="¥#"
        negativePattern="(¥#)"
        zeroValue="¥0"/>
    <JapaneseImperialDateFormat long="EEEE, Gy年M月d日"
        medium="G yy-MM-dd"
        short="yy/MM/dd"
        yearSymbol="年"/>
```

Setting the Japanese Imperial Calendar as the Default for a Locale

ClaimCenter provides a way to set a default calendar for a locale through the `defaultCalendar` attribute of the <GWLore> element. For example, to set the Japanese Imperial calendar (JIC) as the default calendar for the ja_JP locale, add the `defaultCalendar` attribute to the <GWLore> element and set it to JapaneseImperial.

```
<GWLore code="ja_JP" name="Japanese" typecode="ja_JP" defaultCalendar="JapaneseImperial">
```

Working with the Japanese Imperial Date Data Type

Guidewire ClaimCenter provides the ability to create an object extension of type `japaneseimperialdate` based on `java.util.Date`. You can use this data type to do the following:

- To set a calendar field value so that it always displays values in Japanese Imperial calendar style.
- To set a calendar field value so that it displays values in Japanese Imperial calendar style depending on a data field. This requires additional configuration.

For example, consider the following two fields:

- `Policy.EffectiveDate`
- `Policy.ExpirationDate`

Now consider the following cases:

Locale	ClaimCenter displays...
en_US	If the user's locale is en_US, then ClaimCenter is to display these fields using the Gregorian calendar in all situations.
ja_JP	If the user's locale is ja_JP, then ClaimCenter is to display these fields using the Japanese Imperial calendar in all situations.
ja_JP	If the user's locale is ja_JP and Policy.PolicyType!="CALI", then ClaimCenter is to display these fields using the Gregorian calendar.
ja_JP	If the user's locale is ja_JP and Policy.PolicyType=="CALI", then ClaimCenter is to display these fields using the Japanese calendar.

Note: CALI (*Compulsory Auto Liability Insurance*) is a type of automobile policy in Japan that every driver must carry by law. It covers only injury to other parties, basically mandating that every driver protect others around them (but not themselves), just like in the United States. If a driver wants to protect him- or herself, he or she must purchase another policy, known as a Voluntary policy.

Thus:

- In the first two cases, the calendar to use in formatting `EffectiveDate` and `ExpirationDate` is dependant on the locale of the user only. For this, you merely need to set the `defaultCalendar` attribute on `<GWLocale>` (in `localization.xml`) to the correct value.
- In the last two cases, ClaimCenter displays these fields depending on additional conditions. For this, you need to provide additional configuration. See “Setting a Field to Conditionally Display the Japanese Imperial Calendar” on page 545.

See Also

- “Setting the Japanese Imperial Calendar as the Default for a Locale” on page 543
- “Setting a Field to Always Display the Japanese Imperial Calendar” on page 544
- “Setting a Field to Conditionally Display the Japanese Imperial Calendar” on page 545

Setting a Field to Always Display the Japanese Imperial Calendar

Suppose, for example, that you want to add an additional field to the `Activity` object that uses the `japanesempirealdate` data type.

1. First, open `Activity.etx`, which exists in `Data Model Extensions → extensions`.

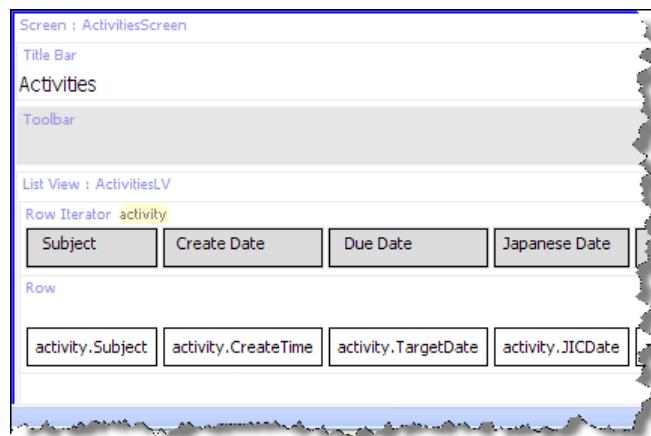
2. Enter something similar to the following:

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="Activity">
  <column desc="For use with Japanese Imperial calendar fields"
    name="JICDate"
    type="japanesempirealdate">
  </column>
</extension>
```

To be useful, you need to use this field to display a value in the Japanese Imperial Calendar format. For example, in the `Activities` screen, you can add a `Japanese Date` field.

Activities			
Subject	Create Date	Due Date	Japanese Date
Approval	07/28/2009	07/28/2009	平成 21/07/15

In the Activities PCF file, it looks similar to the following:



IMPORTANT You must configure your installation for the Japanese locale in order to display a date in Japanese Imperial calendar format. Otherwise, regardless of the calendar setting, you see only Gregorian dates.

Setting a Field to Conditionally Display the Japanese Imperial Calendar

Suppose, for example, that you want to treat a field as either a Gregorian date or a Japanese Imperial calendar date depending on certain factors. In this case, it is possible to write a datatype annotation that causes ClaimCenter to display certain fields in different formats in different situations. For example (as described in “Working with the Japanese Imperial Date Data Type” on page 543), suppose that you want to do the following:

- If the user's locale is ja_JP and Policy.PolicyType!="CALI", then ClaimCenter is to show these fields using the Gregorian calendar.
- If the user's locale is ja_JP and Policy.PolicyType=="CALI", then ClaimCenter is to show these fields using the Japanese calendar.

ClaimCenter provides several different ways to accomplish this date formatting. For example, you can:

- Annotate an entity field (database column)
- Annotate a Gosu property

To annotate an entity field

1. First, configure the data type for the entity field in an extension file. For example:

```
<column name="JICDate" type="japaneseimperialdate"/>
```

See the discussion in “Setting a Field to Always Display the Japanese Imperial Calendar” on page 544 for details if necessary.

2. Define a `PresentationHandler` class for the data type. For example:

```
class JapaneseImperialDateTypeDef implements IDatatypeDef {  
  
    construct() { }  
  
    override property get PresentationHandler() : IDatatypePresentationHandler {  
        return new JapaneseImperialCalendarPresentationHandler()  
    }  
    ...  
}
```

If the class that you create implements an interface (in this case, `IDataTypePresentationHandler`), then Studio highlights the interface name in red and marks the class as invalid. This is because you need to provide implementations for all methods and properties in the interface. If you place your cursor in the class definition line, and press ALT-ENTER, Studio automatically inserts the required method and property bodies for you. You then need to define these properties and methods to suit your business needs.

Note: For an example of how to implement a presentation handler class, see “Defining a New Tax Identification Number Data Type” on page 310.

3. Implement the date presentation handler for the data type. You can make the implementation logic dynamic based on the entity context. For example:

```
class JapaneseImperialCalendarPresentationHandler implements IDatePresentationHandler {  
    construct() {}  
  
    override function getCalendar( ctx: Object, prop: IPropertyInfo ) : DisplayCalendar {  
        /** if the "ctx" object is an policy and the policy is "cali" type and the "prop"  
         * is the effective date field ...  
        **/  
        if (...)  
            return JAPANESEIMPERIAL  
        else  
            return GREGORIAN  
    }  
    ...  
}
```

See “Sample JIC Presentation Handler” on page 546 for sample code for a JIC presentation handler.

To annotate a Gosu property

1. Annotate the data type for the Gosu property. For example:

```
@DataType("japanesempirealdate")  
property get JICDate() : Date {  
    return _bean.DateField1  
}
```

2. Define a `PresentationHandler` class for the data type in a similar fashion to that described previously in step 2 of *To annotate an entity field*.
3. Implement the date presentation handler class for the data type in a similar fashion to that described previously in step 3 of *To annotate an entity field*.

See Also

- For a working example of how to implement the `IDataTypeDef` interface, see “Defining a New Tax Identification Number Data Type” on page 310. Specifically, see “Step 2: Implement the `IDataTypeDef` Interface” on page 311.
- For sample code for a JIC presentation handler, see “Sample JIC Presentation Handler” on page 546.

Sample JIC Presentation Handler

The following sample code illustrates how to create a Japanese Imperial Calendar (JIC) presentation handler. The code returns a JIC date for Japanese citizens. It also returns a JIC data if the loss location is in Japan.

```
package gw.datatype.presentation  
  
uses gw.datatype.handler.IDatePresentationHandler  
uses gw.lang.reflect.IPropertyInfo  
uses gw.datatype.DisplayCalendar  
uses pel.jp.util.PELUtil  
  
class JapaneseImperialCalendarPresentationHandler implements IDatePresentationHandler {  
  
    override function getDisplayFormat(p0 : Object, p1 : IPropertyInfo) : String {  
        return null //## todo: Implement me  
    }  
}
```

```
@Param("ctx","The object displaying the date property")
@Param("prop","Details of the date property")
override function getCalendar(ctx : Object, prop : IPropertyInfo) : DisplayCalendar {
    gw.api.util.Logger.logInfo("~~~ Running overridden getCalendar method")
    gw.api.util.Logger.logInfo("~~~ ctx is: " + ctx)
    gw.api.util.Logger.logInfo("~~~ prop is: " + prop.Name)

    //Only show Imperial birthdates for Japanese citizens
    if(ctx typeis Person and prop.Name == "DateOfBirth" and ctx.Nationality == Country.TC_JP) {
        gw.api.util.Logger.logInfo("~~~ Person.date of type = " + prop.Name)
        return JAPANESEIMPERIAL
    }
    //Show Imperial date if the loss location is Japan
    else if(ctx typeis Exposure and ctx.Claim.LossLocation.Country == Country.TC_JP) {
        gw.api.util.Logger.logInfo("~~~ Imperial date of type = " + prop.Name)
        return JAPANESEIMPERIAL
    }
    else {
        return GREGORIAN
    }
}
```

part VIII

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 551
- “Starting the Debugger” on page 552
- “Setting Breakpoints” on page 552
- “Stepping Through Code” on page 554
- “Viewing Current Values” on page 554
- “Resuming Execution” on page 555
- “Using the Debugger with the GUnit Tester” on page 555
- “Using the Gosu Tester” on page 555
- “Suggestions for Testing Rules” on page 558

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 (unsurprisingly enough) through the **Studio Debug** 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 prompts you to connect to one.) If the debugger is active, you can debug Gosu code that runs in the Gosu Tester 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.

IMPORTANT You must be connected to a running application server to execute queries in Studio or the Studio debugger. Guidewire *expressly does not support* performing debugging operations on a live production server.

Starting the 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 have 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.)

You can start the debugger in one of the following modes:

Debug	Description
 Debug	Debugger starts and pauses at the location specified. Studio pauses only before running particular lines of code that you explicitly specify. Use this mode to set breakpoints at specific lines of code. This enables you to see the point at which Studio invokes the code and you can view the values of variables or object properties at that moment. For example, if a calculation is not giving you the expected result, you can step through its code and see what values the Gosu code actually uses.
 Debug and break	Debugger starts and breaks on the next Gosu call. Studio pauses before running the next Gosu code that it finds. Use this mode to see which line of code ClaimCenter invokes in response to a particular action in ClaimCenter. This enables you to examine the properties of the activity or other entity that are relevant to that code. For example, if you use automated assignment on an activity, Studio pauses before it actually makes the assignment. You can then see which assignment code ClaimCenter is choosing to run. This can help you understand what led ClaimCenter to make the assignment choice that it did.

Choosing either **Debug → Debug only...** or **Debug → Debug and break only...** gives you the following additional choices:

- | | |
|-------------|--|
| Server | Select this option if you want to debug rules called from within the ClaimCenter user interface or Gosu code called from within a PCF page. |
| Gosu Tester | Select this option if you want to debug Gosu code called from within the Gosu tester. (See “Using the Gosu Tester” on page 555 for details.) |

If you start the debugger in any mode, Studio shows the **Debugger** panel. You can view the connected host by opening the **Frame** tab.

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

Use the following **Debug** menu commands to set a breakpoint:

Menu command	Description
Toggle Line Breakpoint	Toggles a breakpoint (either on or off) on the line of code on which the cursor rests. The debugger pauses before executing this line of code.

You can also set a breakpoint by clicking in the gray area next to a code line. Studio then inserts the breakpoint symbol  in that area. (Studio does not permit you to set a breakpoint on code that has been commented out.)

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.

Note: You can set a breakpoint in a rule condition statement as well.

To view a breakpoint

Use the **Debug** menu command **Edit Breakpoints** to view and manage breakpoints. Selecting this menu item opens that **Edit 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 or remove a breakpoint, but not add a breakpoint.

To remove a breakpoint

You remove a breakpoint from your code by clicking the breakpoint symbol  next to the code line. Studio then removes the breakpoint. You can also:

- Select **Debug** → **Remove All Breakpoints** to remove all breakpoints simultaneously.
- Select **Debug** → **Toggle Line Breakpoint** to remove an existing breakpoint from the line in which the cursor resides.
- Select **Debug** → **Edit Breakpoints** and use that dialog to deactivate or remove a breakpoint.

Minimizing the Impact to Users

The debugger responds to the actions of all users, and thus it stops on the actions of the first user who triggers it. As a consequence, unsuspecting users can find their sessions paused while the debugger awaits your attention. To minimize this impact, you can constrain the debugger to respond to actions in a single user account only.

IMPORTANT Guidewire explicitly and strongly recommends that you do **not** perform debugging operations on live production systems. This means that Guidewire *expressly* recommends that you do **not** attempt to connect Guidewire Studio to a live production system and run debug tests on it.

To constrain the debugger

1. Select **Edit User Name...** from the **Debug** menu. This command opens a dialog box containing two fields: **Debugger** and **User Name**.
2. Select the desired Studio debugger from the **Debugger** drop-down list. You have the following choices:
 - Server

- Gunit Test
- Gosu Tester

See the discussion on “Starting the Debugger” on page 552 for information on these choices.

3. Enter a valid user name in the **User Name** field.

4. Click **OK**.

Thereafter, the debugger ignores all calls **except** from that user account. In this manner, you can restrict the impact of running the debugger to a single user account.

To remove the constraint

1. Open the **Edit User Name** dialog box.
2. Select the same **Debugger** type that you did previously.
3. Leave the **User Name** field empty.
4. Click **OK**.

This removes the constraint and resets the debugger to respond to all logged-on users.

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, select **Debug** → **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, select **Debug** → **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, select **Debug** → **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 **Frame** tab of the **Debugger** panel 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 **Frame** 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 **Frame** tab. 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, click the **Watches** tab in the debugger, right-click on an empty part of the tab, and then click **Add Watch**. Type the Gosu expression to add to the list.

Note: You can only access the **Add Watch** option if the Studio debugger has stopped at a breakpoint.

As you step through each line of code in the debugger, keep the **Watches** tab 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:

- **Continue debugging.** Resume normal execution, but pause again at the next breakpoint, if any. To continue debugging, select **Debug → Resume Execution**. 
- **Stop debugging.** Resume normal execution, and ignore all remaining breakpoints. To stop debugging, select **Debug → Stop**. 

Using the Debugger with the GUnit Tester

Guidewire Studio includes a GUnit tester that you can use to configure and run tests of your Gosu code. The Gunit tester works automatically and seamlessly with the embedded QuickStart servlet container, enabling you to see the results of your GUnit tests within Studio.

Selecting **GUnit → Run** (Ctrl-F10) or **GUnit → Debug** (Ctrl-F9) opens the **GUnit Run/Debug Settings** configuration dialog in which you can set the GUnit configuration and perform other similar tasks. For details on working with the GUnit tester, see “Using GUnit” on page 559.

Using the Gosu Tester

You use the Gosu Tester 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 tester and see immediate results.

To run the Gosu tester, select **Gosu Tester** from the Studio **Tools** menu or use the toolbar icon.  It is possible to save a Gosu test expression and open it again in the Gosu tester by using the appropriate menu commands.

Note: *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 find query is always a read-only query object. To work with this object, you must add it to a bundle. See “Using the Results of Find Expressions (Using Query Objects)” on page 164 in the Gosu Reference Guide.*

The Gosu tester 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.

IMPORTANT If you create a code that contain an infinite loop in the Gosu tester, it is not sufficient to shut down the tester to correct the problem. You must also shut down and restart Studio itself. Merely shutting down the Gosu tester is not sufficient to stop the running JVM.

Testing a Gosu Expression

To test an expression in the Gosu Tester, click **Expression** at the top of the Tester window. Type your expression in the **Gosu Source** box, and then click **Run** to evaluate the expression. The result of evaluating the expression is shown in the **Runtime Output** box.

For example, the expression:

```
"Testing " + 1 + 2 + 3
```

produces the output:

```
Return Value: Testing 123
```

Testing a Gosu Program

A Gosu program consists of one or more complete executable Gosu statements. To test a program in the Gosu Tester, click **Program** at the top of the Tester window. Type your program code in the **Gosu Source** box, and then click **Run** to execute the program. The output of the program, if any, is shown in the **Runtime Output** box.

The **Runtime Output** box shows any output explicitly produced by the program, such as output from the `print` statement or exception stack traces. If you call an external program (for example, Java code), anything sent to standard output and standard error is also output.

For example, the following program:

```
var claim : Claim = Claim( 12 );
for( var exposure in claim.Exposures ) { print( exposure.ClaimantType ); }
```

produces the following output:

```
veh_other_owner
insured
householdmember
veh_other_owner
```

Testing a Gosu Template

To test a Gosu template in the Gosu Tester, click **Template** at the top of the Tester window. Type your template code in the **Gosu Source** box, and then click **Run** to process the template. Studio shows the template output in the **Runtime Output** box.

For example, the template:

```
<HTML>
<BODY>
<%
    // This scriptlet declares and initializes a date
    var d = gw.api.util.DateUtil.currentDate()
%>
<%-- Display the date using an expression --%>
Hello! Today's date is: <%= d %>.
```

```
</BODY>
</HTML>
```

produces the output:

```
<HTML>
<BODY>
Hello! Today's date is: August 25, 2006
</BODY>
</HTML>
```

Testing Gosu Classes

Guidewire Studio also provides a mechanism for debugging and testing Gosu classes using the Gosu Tester. This works only with Gosu classes called from code running in the Gosu Tester, not on Gosu classes that the tester is running directly.

To debug Gosu classes

1. Start the debugger using either **Debug** → **Debug** or **Debug** → **Debug and break** and select the **Gosu Tester** debugger.
2. Call the class code that you want to debug from within the Gosu Tester.
3. Test and debug the Gosu code as appropriate.

Generating Testing and Debugging Information

It is often useful to use the Gosu Tester to generate and print information about entities and objects. You can then use this information to test and debug your Gosu code, rules, classes, and enhancements. For example, suppose that you want to generate information about a claim object (in Guidewire ClaimCenter). You could write a small program that outputs information about the claim object and use the Gosu Tester to run the program and generate the output.

The following code sample illustrates how to generate information about an object.

```
var query = find(c in Claim where c.ClaimNumber == "235-53-365870")
var theObject = query.getAtMostOneRow()
var myType = typeof theObject
var sb = new java.lang.StringBuffer()

sb.append(myType).append(" {\n")

for (var p in myType.TypeInfo.Properties) {
    var val : Object
    if (NOT p.Readable) continue

    try {
        val = p.Accessor.getValue( theObject )
    } catch (ex) {
        val = "KABOOM:"+ex.Message
    }

    sb.append(p.Name).append(" <").append(p.Type).append("> := ").append(val).append("\n")
}

sb.append("}")
print(sb)
```

This code generates the following output.

```
//GENERATED OUTPUT
entity.Claim {
    Bundle <gw.transaction.Bundle> := Bundle #17956136 :
    Read      : 0
    Updated   : 0
    Insert    : 0
    Removed   : 0
    Total     : 0
    Beans By Class  :

IntrinsicType <gw.lang.reflect.IType> := entity.Claim
```

```
Changed <boolean> := false
BeanChanged <boolean> := false
ChangedFields <java.util.Set> := []
ChangedFieldsForRow <java.util.Set<gw.entity.IEntityPropertyInfo>> := []
ChangedFieldsIncludingOwnedArrays <java.util.Set> := []
DisplayName <java.lang.String> := 235-53-365870
EntityIntrinsicType <gw.entity.IEntityType> := entity.Claim
OriginalVersion <com.guidewire.commons.entity.Bean> := 235-53-365870
LOCKINGCOLUMN_DYNPROP <com.guidewire.commons.metadata.types.ColumnDynPropertyInfo> := LockingColumn
LOADCOMMANDID_DYNPROP <com.guidewire.commons.metadata.types.ColumnDynPropertyInfo> := LoadCommandID
NewlyImported <boolean> := false
PublicID <java.lang.String> := demo_sample:1
...
```

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 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 551 for details.)

This topic includes:

- “The TestBase Class” on page 559
- “Configuring the Server Environment” on page 560
- “Configuring the Test Environment” on page 563
- “Creating a GUnit Test Class” on page 565
- “Using Entity Builders to Create Test Data” on page 568
- “Running Gosu API Tests” on page 577

IMPORTANT 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

Note: For a complete discussion of how to write Gosu code and work with it, see the *Gosu Reference Guide*. The following discussion provides a summary of important key points only.

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 implementations 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 568 for details on how to create test data.
- See “Creating a Builder for a Custom Entity and Testing It” on page 575 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 (`Tests → New → Test`), 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 559 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>@Archive</code>	Upgrades or restores the archive database after the server starts.
<code>@ChangesCurrentTime</code>	Sets up a mock system clock that allows the test to change the current time during the test.
<code>@IncludeModules</code>	Allows specification of custom configuration modules to include in the test server. You can specify an array of directories as the additional configuration roots. By default, Studio uses the “test-config” directory (relative to the test) as the default directory if you do not specify a directory name. See “The <code>@IncludeModules</code> Annotation” on page 562 for more information.
<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.

Annotation (@gw.testharness.*)	Description
@RunLevel	Allows a test to run at a different run level. The default value is RunLevel.NO_DAEMONS. You can, however, change the run level to one of the following (although each level takes a bit more time to set up): <ul style="list-style-type: none"> • RunLevel.NONE - Use if you do not want any dependencies at all. • RunLevel.SHUTDOWN - Use if you want all the basic dependencies set up, but with no database connection support. • RunLevel.NO_DAEMONS - Use for a normal server startup without background tasks. (This is also suitable for SOAP tests.) • RunLevel.MULTIUSER - Use to start a complete server (batch process, events, rules, Web requests, and all similar components).

The @IncludeModules Annotation

IMPORTANT Studio does not permit you to make data model changes using the @IncludeModules annotation. This means that you cannot make new entity or typelist extensions, for example, through an included configuration file.

Use the @IncludeModules test annotation to declare that a test is to be run with additional configuration modules. For example, suppose that you want to run a test with an alternate value for a display key. The name of this test is gw.demo.testMyClass, and you store it at the following location:

```
../modules/configuration/resources/tests/gw/demo/MyTest.gs
```

To set up the test module:

1. First, create a module root in the same directory as the test source and call it **test-config**.
2. Then, create a custom display properties file and place it in the correct location under the module root:

```
locale/en_US/display.properties
```

The full path to the custom display properties file would be the following:

```
../modules/configuration/resources/tests/gw/demo/test-config/locale/en_US/display.properties
```
3. Finally, annotate the test class with the @IncludeModules annotation.

The code would look similar to the following:

```
package gw.demo

uses gw.testharness.IncludeModules
uses gw.testharness.TestBase

@ServerTest
@includeModules
public class MyTest extends TestBase {
    ...
}
```

As you run the test, Studio pulls the display key value from the **display.properties** file under the **test-config** directory root.

If you would like to use a directory other than the default **test-config** directory, you can specify an argument to @IncludeModules. For example:

```
package gw.demo

uses gw.testharness.IncludeModules
uses gw.testharness.TestBase

@ServerTest
@includeModules({"my-test-config"})
public class MyTest extends TestBase {
    ...
}
```

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:

- By selecting **Debug** from the Studio GUnit menu.
- By selecting **Run** from the Studio GUnit menu.
- By selecting an existing GUnit test using the Studio toolbar  icon.
- By selecting an existing GUnit test class (in the **Resources** tree) and using the right-click menu command **Run ClassTest**.

Using this dialog, you can:

- Set various default configuration parameters for all tests.
- Add a test configuration to the list (at the left) and configure parameters specifically for that particular test.

To set 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, select the **GUnit** node from the panel at the left. You see **Default configuration for all GUnit tests** at the top of the dialog. Enter the default configuration parameters as appropriate. See “Configuration Parameters” on page 563 for a description of the various configuration parameters.

To add 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 **GUnit** in the left-side panel of the dialog. Use the following dialog toolbar icons to work the configuration in the left-hand panel.

Icon	Use to
	Add a new named test configuration to the list
	Delete the selected configuration from the list
	Clone the selected test configuration
	Shift the selected configuration up within the list
	Shift the selected configuration down within the list

To view the configuration settings before launching. It is possible to turn on, or off, the **Run/Debug Settings** dialog before running a test. To view the GUnit configuration settings before launching a test, you must check the **Display settings before launching** option in the bottom left-hand corner of the **Run/Debug Settings** dialog.

If you uncheck this option, you do not see the **Run/Debug Settings** dialog upon starting a test. (To start a test, you select it from the resource tree, and then choose the **Run** command from the right-click menu that corresponds to your test.) Instead, the test starts immediately. In addition, selecting **Run** or **Debug** from the Studio GUnit menu does not open this dialog either. To access the **Run/Debug Settings** dialog again, choose **Edit Configurations...** from the list of choices available from the Studio toolbar  icon.

Configuration Parameters

Use the **Run/Debug Settings** dialog to enter the following configuration parameters:

- Name
- Test Type
- JVM Options
- Java Debug Options

- Module

Note: 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 565 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 the **Add** icon  and create a new blank configuration.
- Select an existing configuration, then click the **Clone** icon  to copy the existing configuration parameters to the new configuration.
- Select the test class in the **Resources** tree and right-click and select **Run ClassTest**. Then, select the name of the test from the list of GUnit tests and click the **Save** icon. 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 Type

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 *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.

JVM 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 Parameters** text field. 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.

You also have the option of using a Java debugger while you run and debug code. The Java debugger provides inspection and debugging of a local or remote Java Virtual Machine. (It does this using the JPDA—Java Platform Debugger Architecture—libraries).

- To open the Java debugger during test execution, select **Run Java debugger when running tests**.
- To open the Java debugger for use in debugging tests, select **Run Java debugger when debugging tests**.

Java Debug Options

These include:

- *Transport* - Use to specify the type of communication to use with the Java debugger. Transport defines the type of communication to use between the debugger application process and the virtual machine to debug (the target VM). Your choices are:

Socket Socket transport uses standard TCP/IP sockets to communicate information between the Java debugger application and the target VM.

If you select **Socket**, enter the appropriate information in the **Host** and **Port** fields.

- The **Host** field specifies the host on which the remote process runs and to which the debugger connects.
- The **Port** field specifies the socket port on the host to which the debugger connects.

Shared Memory Shared memory transport addresses are simply names that can be used as Win32 file-mapping object names. If you select **Shared Memory**, enter the appropriate information in the **Shared memory address** field.

- *Suspend startup* - If you select **Yes**, the debugger suspends the target VM immediately before the loading the main class.

Module

Use to set which module to run this configuration against.

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 **Tests** folder in the **Resources** tree. 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 560 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”. (JUnit 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. Expand the Studio Resources tree and select the Tests node.
2. Select New → Package from the right-click menu. (This is your only choice.) Continue creating your directory (package) structure as necessary.
3. Select your new package node, right-click, and select New → Test.
4. Enter the test class name in the New Test Class dialog. 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. Select the test class that you want to run or debug from the Resources → Classes folder.
2. Right-click and select either Run `myTestClass` or Debug `myTestClass` from the menu. This action opens a test console at the bottom of the screen.
 - The left-hand pane displays your test class as a root node and the individual methods as child nodes.
 - The right-hand pane displays the console log, which displays information about the test and provides error or failure messages if any given test fails.If a test succeeds, Studio displays a check mark icon next to the method (test) name in the left-hand pane. If a test fails, Studio displays an X check mark next to the method name in the left-hand pane. If all tests succeed, Studio turns the test progress bar green. If one or more tests fail, Studio turns the test progress bar red.
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 563.

Using Entity Builders to Create Test Data

IMPORTANT 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()
```

Note: For readability, Guidewire strongly 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 <i>not</i> 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 <i>and</i> 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.

Note: GUnit 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.

Note: If you do not use this method inside a transaction block that automatically commits a bundle, then you must commit the bundle yourself. To do so, add `bundle.commit` to your code.

Entity Builder Examples

The following examples illustrate various ways that you can use builders to create sample data for use in GUnit tests.

- Creating Multiple Objects from a Single Builder
- Nesting Builders
- Overriding Default Builder Properties

Creating Multiple Objects from a Single Builder

The Builder class creates the builder object at the time of the `create` call. Therefore, you can use the same builder instance to generate multiple objects.

```
var activity1 : Activity
var activity2 : Activity
var bundle = gw.transaction.Transaction.runWithNewBundle( \ bundle -> {
    var activityBuilder = new gw.api.builder.ActivityBuilder()
        .withType( "general" )
        .withPriority( "high" )
    activity1 = activityBuilder.withSubject( "this is test activity one" ).create( bundle )
    activity2 = activityBuilder.withSubject( "this is test activity two" ).create( bundle )
} )
```

Nesting Builders

It is possible to nest one builder inside of another by having a method on a builder that takes another builder as a parameter. 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 myTestOf{
    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 572
- “Extending the DataBuilder Class” on page 573
- “Creating a Builder for a Custom Entity and Testing It” on page 575

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
    }

}
```

Note: 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 568):

Initial word	Indicates
<code>on</code>	A link to a parent, for example, <code>PolicyPeriod</code> is on an <code>Account</code> , so the method is <code>onAccount(Account account)</code> .
<code>as</code>	A property that holds only a single state, for example: <code>asBusinessType</code> or <code>as AgencyBill</code> .
<code>with</code>	The single element or property to be set. For example, the following sets a <code>FirstName</code> property: <code>withFirstName("Joe")</code>

Your configuration methods can set properties by calling `DataBuilder.set` and `DataBuilder.addArrayElement`. You can provide property values as any of the following:

- Simple values.
- Beans to be used as subobjects.

- Other builders, which 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 implementing builder configuration methods. This enables callers of your builder to select the execution order through the order of the configuration method calls.

Constructors for builders can call `set`, and similar methods to set up default values. These are useful to satisfy `null` constraints so it is possible to commit built objects to the database. However, Guidewire generally recommends that you limit the number of defaults. This is so that you have the maximum control over the target object.

Other DataBuilder Classes

The `gw.api.databuilder` package also includes `gw.api.databuilder.ValueGenerator`. You can use this class, for example, to generate a different value for each instance constructed to satisfy uniqueness constraints. The `databuilder` package includes `ValueGenerator` class variants for generating unique integers, strings, and type-keys:

- `gw.api.databuilder.IntegerStringGenerator`
- `gw.api.databuilder.SequentialStringGenerator`
- `gw.api.databuilderTypekeyStringGenerator`

Custom Builder Populators

Ideally, all building can be done through simple property setters, using the `DataBuilder.set` or `DataBuilder.addArrayElements` methods. However, you may want to define more complex logic, if these methods do not suffice. To achieve this, you can define a custom implementation of `gw.api.databuilder.populator.BeanPopulator` and pass it to `DataBuilder.addPopulator`. Guidewire provides an abstract implementation, `AbstractBeanPopulator`, to support short anonymous `BeanPopulator` objects.

The following example uses an anonymous subclass of `AbstractBeanPopulator` to call the `withCustomSetting` method. This code passes the group to the constructor, and the code inside of `execute` only accesses it through the `vals` argument. This allows the super-class to handle packaging details.

```
public MyEntityBuilder withCustomSetting( group : Group ) {  
    addPopulator( new AbstractBeanPopulator<MyEntity>( group ) {  
        function execute( e : MyEntity, vals : Object[] ) {  
            e.customGroupSet( vals[0] as Group )  
        }  
    } )  
    return this  
}
```

The `AbstractBeanPopulator` class automatically converts builders to beans. That is, if you pass a builder to the constructor of `AbstractBeanPopulator`, it returns the bean that it builds in the `execute` method. The following example illustrates this.

```
public MyEntityBuilder withCustomSetting( groupBuilder : DataBuilder<Group, ?> ) : MyEntityBuilder {  
    addPopulator( new AbstractBeanPopulator<MyEntity>( groupBuilder ) {  
        function execute( e : MyEntity, vals : Object[] ) {  
            e.customGroupSet( vals[0] as Group )  
        }  
    } )  
    return this  
}
```

```

        }
    }

    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
<code>ExtSystem</code>	<code>Typekey</code>
<code>UserName</code>	<code>String</code>
<code>Password</code>	<code>String</code>

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.etx</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 `Typelists`, right-click and select `New → Typelist`. Add a few *external system* typecodes. (For example, add `SystemOne`, `SystemTwo`, or similar items.)

2. Create `ExtCredential`. Select the `extensions` folder, right-click and select `New → Other file`. Name this file `ExtCredential.eti` and enter the following:

```

<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel" entity="ExtCredential" table="extcred"
         type="retireable" exportable="true" platerform="true" >
    <typekey name="ExtSystem" typelist="ExtSystemType" desc="Type of external system"/>
    <column name="UserName" type="shorttext"/>
    <column name="Password" type="shorttext"/>
    <foreignkey name="UserID" fkentity="User" desc="FK back to User"/>
</entity>

```

3. Modify `User`. Find `User.etx` (in `extensions`). If it does not exist, then you must create it. However, most likely, this file exists. Open this 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 238 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 573 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 “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. (JUnit 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 183 in the *Gosu Reference Guide*.)
- It uses a `beforeClass` method to create the `ExtCredential` test data. This method calls `ExtCredentialBuilder` as part of a transaction block, which creates and commits the bundle automatically. GUnit calls the `beforeClass` method before it instantiates the test class for the first time. Thereafter, the test class uses the test data created by the `beforeClass` method. It is important to understand that GUnit does not drop the database between execution of each test method within a test class. However, if you run multiple test classes together (for example, by running all the test classes in a package), GUnit resets the database between execution of each test class.
- It defines several test methods, each of which starts with `test`, with each method including an `assertXXX` method to test the data.

If you run the `ExtCredentialBuilderTest` class as defined, the GUnit tester displays green icons, indicating that the tests were successful:

Running Gosu API Tests

Note: In general, to test an API, Guidewire recommends that you test the API contract, not the API implementation details.

You can create GUnit tests in Studio that test Gosu APIs. To test, Studio runs the API test in a server and communicates with Axis using a Web server to exercise all the standard Axis code paths. Guidewire exposes the server

APIs to tests under the `soap.local` namespace, which is split up in the same fashion as the generated Java entities.

Namespace	Description
<code>soap.local.api.*</code>	SOAP service objects
<code>soap.local.enumeration.*</code>	SOAP simple types (for example, typekeys)
<code>soap.local.entity.*</code>	All SOAP exposed entities (for example, the Claim entity)
<code>soap.local.fault.*</code>	All SOAP faults

IMPORTANT Use the `soap.local` name space for testing on your local machine **only**. Do not use these classes for web services in a production environment. If you choose to use the `soap.local` namespace, then you must add the `@SOAPLocalTest` annotation to your test code. If you do not supply this annotation, ClaimCenter does not load the soap local typeloader into the type system. See the sample code in “SOAP Local Test Example” on page 578 for an example use of this annotation.

Server Authentication

You perform server authentication using an authentication handler. For example:

```
var adminTools = new soap.local.api.ISystemToolsAPI()
adminTools.addHandler( new GSAuthenticationHandler("username", "password") )
```

See the following example for an illustration on how to use server authentication.

SOAP Local Test Example

If you use the `soap.local` namespace for testing purposes, then you must add the following annotation before the test class declaration:

```
@SOAPLocalTest
```

The following example illustrates the use of the `SOAPLocalTest` annotation. It also illustrates the use of an authentication handler.

```
package MyTests
uses gw.testharness.SOAPLocalTest

@SOAPLocalTest
class IUserAPITest extends gw.testharness.TestBase {

    construct() {}

    function testUserAPI() {
        var api = new soap.local.api.IUserAPI()
        api.addHandler(new gw.api.soap.GWAuthenticationHandler("username", "password"))
        var pid = api.findPublicIdByName("username")
        assertNotNull(pid)
        var user = api.getUser(pid)
        assertNotNull(user)
        assertEquals(user.Credential.UserName, "username")
    }
}
```

Note: For the test to complete successfully, you must replace the `username` and `password` parameters in the code with valid system values.

API Test Example

The following example API test throws a `ServerStateException`. The server throws an exception because the test method tries to set the system run level to the same value to which it is already set.

```
package MyTests

uses gw.testharness.ServerTest
uses gw.api.webservice.exception.SOAPException
uses gw.api.webservice.systemTools.SystemRunlevel
```

```
uses gw.webservice.systemTools.ISystemToolsAPI
uses gw.api.webservice.systemTools.SystemRunlevel

@gw.testharness.ServerTest
class SOAPTest extends gw.testharness.TestBase {
    construct() {
        //Perform initialize here
    }

    function testTwoMaintenanceModesThrowsExceptions() {

        var adminTools = new ISystemToolsAPI()
        //Setting the system run level to Maintenance
        adminTools.setRunlevel(SystemRunlevel.MAINTENANCE)

        try {
            //Set the run level to the same value that it already has
            adminTools.setRunlevel(SystemRunlevel.MAINTENANCE)

            //If, for some reason, the test did not fail, then generate the following error message
            fail("Exception not thrown.")

        } catch (e : SOAPException) {
            //Exception caught and handled

        } finally {
            //Resetting the system run level
            adminTools.setRunlevel(SystemRunlevel.MULTIUSER)
        }
    }
}
```

Running this test produces red icons, indicating failure. This is by design. The test provides an error message for the `ServerStateException`. This message (Server already at runlevel 40) is exactly what one expects to see. The test tries to set the `SystemRunLevel` to a specific value, fails because it is already set to that value, and generates the error.

Guidewire ClaimCenter Configuration

Using the Lines of Business Editor

Note: For more information on how ClaimCenter uses the lines of business typelists, see “Configuring Your Lines of Business” on page 307 in the *Application Guide*.

This topic discusses the Guidewire ClaimCenter lines of business.

This topic includes:

- “Lines of Business in Guidewire ClaimCenter” on page 583
- “The Studio Lines of Business Editor” on page 584
- “Managing References to the LOB Typelists” on page 587

Lines of Business in Guidewire ClaimCenter

A line of business (LOB) is a business unit that is independent of other business units in a company. A key feature of ClaimCenter is its ability to model the responsibilities of each business unit of an insurance carrier. In this way, you model the business structure of your carrier, and tell ClaimCenter how to customize its displays and methods of handling claims for each part of your business.

You accomplish this modeling by a series of typelists that do the following:

- Define what each LOB does, including the policies it writes and the coverages each contains
- Relate the LOBs to each other by the types of losses each covers
- Associate ClaimCenter exposures with coverages

There are six special typelists that define the Lines of Business model in ClaimCenter. The six typelists have a somewhat hierarchical relationship, so Studio represents them in a tree structure. The following table lists the six typelists and their parent-child relationships.

TypeList	Parent	Child
LossType	-	LOBCode
LOBCode	LossType	PolicyType

TypeList	Parent	Child
PolicyType	LOBCode	CoverageType
CoverageType	PolicyType	CoverageSubtype
CoverageSubtype	CoverageType	ExposureType
ExposureType	CoverageSubtype	-

Studio represents the parent-child relationship by adding the child typecode as a category on the parent typecode. It also adds the parent typecode as a category on the child typecode. Studio updates the parent-child relationships as you work in the Lines of Business editor. The editor updates the relationships behind the scenes whenever you add a child to, or remove a child from, its parent.

The LOB hierarchy is not a strict hierarchy. Some relationships in the structure are many-to-many which means some typecodes can have multiple parents. (For example, typecodes from PolicyType, CoverageType, and ExposureType can have multiple parents. In contrast, typecodes from CoverageSubtype and LOBCode can have only one parent.)

If a typecode has multiple parents, it appears under each parent. However, it is important to understand that each appearance represents the same typecode and that editing the typecode in one place affects all appearances of it.

For example, suppose the following:

- You add the same child, say child **C** (PolicyType Additional PIP) to two different parents—**A** (LOBCode Auto-Commercial Auto) and **B** (LOBCode Auto-Personal Auto).
- Then, you add a child **D** (CoverageType EvenMorePIP) to child **C** (PolicyType Additional PIP) of parent **A** (LOBCode Auto-Commercial Auto).

After these changes, you also see that same child (CoverageType EvenMorePIP) on child **C** (PolicyType Additional PIP) of the second parent **B** (LOBCode Auto-Personal Auto).

The Studio Lines of Business Editor

Studio displays the Lines of Business editor in the **Resources** tree. After you expand this folder, you see the six special LOB typelists. Selecting a typelist opens the LOB editor on the right, rooted at the typelist selected.

Note: Studio displays these typelists **only** in the Lines of Business editor. They do not appear in the Typelist editor. This editor is the only place in which you can edit these typelists. For a discussion of the LOB typelists, see “Lines of Business in Guidewire ClaimCenter” on page 583.

After you select a typelist (in the **Resources** tree), Studio displays the hierarchical relationships in the tree in the middle. The root of this tree is simply the typelist that you selected. The next level shows all typecodes in this typelist. If you expand past this level, Studio displays the children of each typecode, which are typecodes from the next typelist.

For example, if you select CoverageType in the **Resources** tree, then CoverageType becomes the root in the middle pane. The second level comprises all CoverageType typecodes and the children of these typecodes are then CoverageSubtypes. You can navigate all the way down to ExposureType.

Studio displays retired codes as grayed out. Studio displays each code with the code **Name** field.

The Context Menu

You can right-click in the middle pane to view additional editing options.

- Add New
- Add Existing
- Remove From Parent

- Delete Typecode
- Export Branch

Add New

Use the **Add New** menu option to add a new typecode. Selecting this option opens a dialog that you use to enter values for the new typecode. If you right-click at the root, this option is always available and allows you to add a typecode to the selected typelist. At other levels, the option is only available if it is possible for the typecode to have more children. (The parent-child relationship for some typecodes is one-to-one. In this case, if there is already a child, you cannot add another.) If you select this option at other levels, you can add a new child to the selected typecode, for example. This means that the editor creates a new typecode in the next level down typelist, and 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 doing this in the editor.

Add Existing

Use the **Add Existing** menu option to add an existing code as the child of the selected code. This option is not available at the root level. At other levels, it is available only if it is possible for the parent to have more children *and* if children exist to add. In some cases, the relationship is many-to-one, in which case a child can only be added to *one* parent. (If there are no children left without a parent, Studio disables the **Add Existing** option.) If you select this option, Studio adds a new blank entry to the child list table on the right. The table is made visible and the cursor put in the correct place so you can start editing the name of the code to add. For more details, see “Child List Table” on page 586.

Remove From Parent

This menu option is available for any level below the list of all typecodes. It is unavailable at the root since it has no parent, and unavailable at the first level since there is no parent relationship defined there. This option removes the selected typecode from the list of children for the parent, but otherwise leaves the typecode as is. (This means that it removes the category of this code on the parent and of the parent code on this one.) It is important to understand that this **might** orphan the selected typecode. Orphaned codes are visible in the tree rooted at the code's list.

Delete Typecode

Use this menu option to delete the selected typecode entirely. It is available at all levels except the root. If you select this option, Studio provides a warning message asking you to confirm the deletion. If confirmed, Studio deletes the typecode from the selected typelist and cleans up any parent or child relationships it has. This means removing categories that reference the code in its parent typelist and in its child typelist. However, Studio does **not** modify other categories in other lists that reference this code.

IMPORTANT This has the potential to create orphan typecodes, if children of the removed code have no other parent.

Export Branch

After selecting **Export Branch**, you can then select whether to export the selected LOB typecode hierarchy as HTML or as CSV. These two options, available at all levels, export the tree to an HTML file or a csv file. Studio exports the tree in tabular form. To print the tree, export it to HTML and use a web browser to print it out.

Editing a Typelist

With the root selected in the middle pane, you can edit properties of the typelist itself. On the right, there is a pane in which you can edit the typelist's filters. This pane acts in a similar fashion to the **Filters** pane in the Typelist editor. You can add, remove, and edit filters in this location.

Editing a Typecode

If you select a typecode in the middle pane, Studio displays the details for that code on the right-hand side and allow you to edit them. At the top of the pane, you see the typecode's name and the typelist to which it belongs. Underneath, there is a section in which you can edit fields specific to the typecode, for example, the code, name, description, priority, and retired fields.

- If you edit the name, Studio updates the tree.
- If you edit the code, Studio updates all typecodes in LOB that point to this one.

In the case of ExposureTypes, you can also edit the IncidentType in the main panel. This edits the category on the ExposureType that points to an incident type. (The IncidentType list is not directly available in the **Categories** section of an ExposureType code.)

The Studio LOB Editor Tabs

The lower tabs contain a number of list tables:

- Child List Table
- Parent List Table
- Categories List Table
- Localizations List Table

Child List Table

You can view the children of this typecode in the lower tab called **Children**, which also specifies from which typelist the children come. Within this lower tab, you can select a child typecode and change it by entering a different code. This removes the other code as a child and add the newly specified one. Until you specify an existing, eligible code, the old one remains. If you change focus from the editing area, then the old one appears again. After you specify a legitimate code, Studio removes the old one. You can either type in the name or select it from the drop-down, as with other Studio tables. You can also use the function buttons to do the following:

- Add an existing typecode (**Add**)
- Add a new type code (**Add New**)
- Remove a typecode (**Remove**)

You cannot, however, ever duplicate a typecode. (The editor never enables this option.)

Parent List Table

The values in the Parent list table are not modifiable. This table merely shows all the parents that have this code as a child.

Categories List Table

This list table shows the codes categories and is editable exactly as the categories in the Typecode editor are. Studio does not display categories for the parent and child relationships in this table.

Localizations List Table

This list table shows the localizations for this code. What you see in this location is exactly the same as that shown in the **Localizations** tab in the Typelist editor.

Referential Integrity

As you make edits in the LOB editor, Studio maintains referential integrity within the parent-child relationships in the LOB structure. This means that if you make a change to a typecode's code or if you delete a typecode, then Studio also updates the typecodes that reference that one. However, outside the LOB structure, this is not true. Therefore, if **CostType** refers to a typecode that you deleted, it still refers to that code. If you save your work,

shut down Studio, and try to restart it, Studio generates an error. (You can still restart Studio and correct the problem, however.)

Saving

After you change a typelist, Studio enables the **Save** button in the main toolbar. Whenever you select a different resource in the far left tree (including navigating to a different LOB typelist), Studio saves all edited typelists in the LOB configuration automatically. Studio also performs an automatic save if you select a different node in the middle-pane tree.

Managing References to the LOB Typelists

As discussed in “Lines of Business in Guidewire ClaimCenter” on page 583, the Studio **Lines of Business** editor manages the following typelists, together with the filtering relationships between these types:

- LossType
- PolicyType
- LOBCode
- CoverageType
- CoverageSubtype
- ExposureType

If you modify one of these typelists through the Studio editor, it is possible that you still need to modify other typelists or PCF files that reference your modified typelist. The **Lines of Business** editor *does not* edit other typelists, Gosu code, and PCF files that reference these six typelists. In some cases, this can lead to error states. In particular:

TypeList	Filtered by...
LossParty	CoverageSubtype
CostCategory	CoverageSubtype
LineCategory	
InsuranceLine	PolicyType
ClaimantType	LossType
LossCause	
MetroReportType	
OfficialType	
PriContributingFactors	
QuickClaimDefault	
ResolutionType	
SeverityType	

A number of PCF files are modal and based on the `LossType` and `ExposureType` typelists. A *modal PCF file* is one in which the screen file that references the page calls the appropriate detail or list view based on the `LossType` or `ExposureType` typecode. The following table lists the modal PCF pages:

TypeList	PCF
<code>LossType</code>	<code>LossDetailsDV</code> <code>NewClaimLossDetailsDV</code> <code>ClaimPolicyGeneral</code> <code>ClaimEvaluationDetailsDV</code> <code>NewClaimLocationsLV</code> <code>NewClaimPolicyGeneralPanelSet</code> <code>PolicyGeneralPanelSet</code> <code>LocationDetailPanelSet</code> <code>LocationsLV</code> <code>LocationDetailPanelSet</code> <code>SubrogationMainDV</code> <code>SubrogationFinancialsDV</code> <code>FNOLWizard_BasicInfoScreen</code> <code>FNOLWizard_NewLossDetailsScreen</code> <code>FNOLWizard_AssignSaveScreen</code>
<code>ExposureType</code>	<code>NewClaimExposureDV</code> <code>ExposureDetailDV,</code> <code>NewExposureDV</code>

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` typeplist.
2. Create the supporting detail and list views in the ClaimCenter interface.
3. Update any external typelist that the `LossType` typekey filters. (This include the `ClaimantType`, `LOBCode`, and `LossCause` typelists.)

Creating the Necessary Detail and List Views

You define both the detail views (DV) and list views (LV) within individual PCF files. Claim and policy-related screens within ClaimCenter can then reference these files. As mentioned, the PCF files that you need to add are modal. This means that 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 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 *Cargo* would have a corresponding file named:

`NewClaimWizardLossDetailsDV.Cargo`

For each typecode that add, you need to create the following claim-related files within the **Page Configuration (PCF)** editor in the **Studio Resources** tree. Replace `<Code>` with the name of your typecode using the naming convention described earlier. Put each file in the listed folder, which is a subfolder of the `claim` folder.

Folder	File	Supports
<code>lossdetails</code>	<code>LossDetailsDV.<Code></code>	Editing existing claims

Folder	File	Supports
FNOL	NewClaimLocationsLV.<Code> NewClaimLossDetailsDV.<Code> NewClaimPolicyGeneralPanelSet.<Code> QuickClaimDV.<Code>	New Claim wizard
planofaction	ClaimEvaluationDetailsDV.<Code>	Creating and editing claim evaluations
policy	LocationDetailPanelSet.<Code> LocationsLV.<Code> PolicyGeneralPanelSet.<Code> PolicySummaryGeneralDV.<Code>	Policy edits and searches
snapshot → <version>	ClaimSnapShotGeneral<version>.<Code>	Claim snapshots

Adding a New *Exposure Type* Typecode

For each new *exposure type* that you define, you must also do the following:

1. Add the new typecode to the `ExposureType` typelist.
2. Create the supporting detail views in the ClaimCenter interface.
3. Update any external typelists that contain the `ExposureType` filters. (This includes the `LossPartyType`, `CoverageType`, and `SeverityType` typelists.)

Creating the Necessary Detail Views

You define both the detail views (DV) within individual PCF files. Claim and exposure-related screens within ClaimCenter can then reference these files. As mentioned, the PCF files that you need to add are modal. This means that 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>`

As with the *loss type* typecode files, 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 add, you need to create the following claim- and exposure-related files within the **Page Configuration (PCF)** editor in the Studio Resources tree. Replace `<Code>` with the name of your typecode using the naming convention described earlier. Put each file in the listed folder, which is a subfolder of the `claim` folder.

Claim folder	File	Supports
exposures	ExposureDetailDV.<Code>	Editing existing exposures
newclaimwizard	NewClaimExposureDV.<Code>	New Claim wizard
newexposures	NewExposureDV.<Code>	New Exposure wizard
snapshot → <version>	ClaimSnapShotExposure<version>DV.<Code>	Claim snapshots

Configuring Policy Behavior

There are several aspects of policy behavior that you can configure in ClaimCenter.

This topic includes:

- “Managing Aggregate Limits” on page 591
- “Specifying the Subtabs on a Policy” on page 596
- “Defining Internal (ClaimCenter-only) Policy Fields” on page 597

Managing 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. Typically, insurers use the aggregate cap:

- Regardless of the number of claims made against the relevant policies
- Regardless of the number and variety of exposures represented in the claims

The management tasks associated with aggregate limits include:

- Specifying the financial transactions that count against a limit
- Configuring policy period definitions
- Recalculating (manually) how much of your aggregate limits have already been used

Note: If you want to use ClaimCenter to track and display aggregate limits, add the **Aggregate Limit** subtab to the relevant policy types. For more information, see “Specifying the Subtabs on a Policy” on page 596.

Understanding Aggregate Limits

An aggregate limit can apply to a policy, a specific coverage (or group of coverages), 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 of the policies belonging to the account.

Policy periods play an important role in aggregate limits. Within ClaimCenter, the application uses policy periods to do the following:

- *To connect aggregate limits to either accounts or individual policies.* ClaimCenter associates aggregate limits with policy periods and policy periods identify the claims to which the aggregate limit applies.
- *To distinguish between policy versions.* Policies are typically in effect for a single year (or portion thereof). Each year a policy is in effect is a different version of the policy, with different effective dates.

For more about policy periods, see the section “Configuring Policy Periods” on page 593.

There are two types of aggregate limits: **Loss date** and **Reported date**. For either type, a claim applies to the limit if the claim's date is on or within its policy's effective and expiration date. For **Loss date** aggregate limits the loss date must fall within the effective dates of the policy. For **Reported date** limits, the reported date must be within the effective dates.

You can define aggregate limits on the **Aggregate Limits** tab of a claim's **Policy** page.

- If the aggregate limit grouping is of type **Account**, then any aggregate limit defined on that page applies to the account as a whole, rather than to the individual policy.
- If the aggregate limit grouping is of type **Policy**, aggregate limits defined on that policy's **Aggregate Limits** tab apply only to that policy.

The **AggregateLimit** entity has the following key properties:

Entity Property	Description
LimitAmount	The amount of the aggregate limit.
LimitType	The aggregate limit type, either loss date or reported date.
PolicyPeriodID	The policy period to which the aggregate limit belongs.
CoverageLines	An aggregate limit can have multiple coverage lines or none at all.
ValueType	Limit or deductible.

Define What Counts Against an Aggregate Limit

ClaimCenter reports how much of an aggregate limit has been used. For each policy type that uses aggregate limits, you must tell ClaimCenter which cost types to count against the limit. Within Guidewire Studio, you define these costs in file `aggregateLimitUsed-config.xml`, located in the **Resources** tree in the **Other Resources** folder. The file consists of a main `AggregateLimitUsedConfig` element and one or more `LimitUsedDef` subelements.

The `LimitUsedDef` element definition has the following format:

```
<LimitUsedDef>
  <AggLimitPolicyType code="policy_type"/>
  <AggLimitCostType code="cost_type" calctype="calctype"/>
</LimitUsedDef>
```

The `LimitUsedDef` element defines the cost types that count against the limits of particular policies. You can create multiple `LimitUsedDef` elements to associate different policies with different cost types. You must specify at least one. This element contains two subelements `AggLimitPolicyType` and `AggLimitCostType`.

- 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.xml` typelist defines the list of possible types.
- The `AggLimitCostType` subelement specifies the cost types and financial calculations that count against a limit. The `code` attribute specifies a cost type from the `CostType.xml` typelist. A cost type can appear at most one time in an individual `LimitUsedDef` element.

The `calctype` attribute defines the calculation to apply while determining the used 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.

In the following example, the `LimitUsedDef` element defines the costs for commercial automobile policies. For these policies, the total incurred value of transactions for both **Claim Cost** and **Expense A&O** are subject to the aggregate limit:

```
<LimitUsedDef>
  <AggLimitPolicyType code="auto_comm"/>
  <AggLimitCostType code="aoexpense" calctype="TotalIncurred"/>
  <AggLimitCostType code="claimcost" calctype="TotalIncurred"/>
</LimitUsedDef>
```

From this, ClaimCenter calculates an aggregate limit's limit used 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.
- 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.

Configuring Policy Periods

A policy period identifies the policy or policies that count against 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 against an aggregate limit

A *policy period definition* is a configuration object that determines the form and functionality of one or more policy periods. ClaimCenter creates policy periods as needed for aggregate limits, based on the policy period definitions in the `policyperiod-config.xml` file. (Studio locates this file in **Other Resources**.) The definitions in this file function as the blueprint for your policy periods.

ClaimCenter bases aggregate limits on policy periods. A specific and unique policy period identifies each claim that counts against an aggregate limit. Any time that it creates a policy, 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>
```

The `PolicyPeriodDef` element defines an individual policy period. The `type` parameter takes one of two values: `policy` or `account`. 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 `PolicyTypeConfig` element specifies the policy type to which the definition applies. This is a code from the `PolicyType.xml` 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.

Account	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 alphanumerical value that you append to a policy number to differentiate between effective years for a policy. Typically, you use this only for policy periods of type <code>policy</code> .

Example of Policy-based Period Definition

The following `PolicyPeriodDef` example ensures that all claims made against a specific automobile policy count against a single aggregate limit.

```
<PolicyPeriodDef type="policy">
  <PolicyTypeConfig code="auto_comm"/>
  <PolicyTypeConfig code="auto_per"/>
  <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

The `PolicySuffix` field acts as an implicit time period. It identifies both an effective date and an expiration date. By including the suffix, you insure that claims made in different years are not mistakenly summed into the same limit.

Example of Account-based Period Definition

The following `PolicyPeriodDef` example ensures one aggregate limit applies to all policies belonging to the same account.

```
<PolicyPeriodDef type="account">
  <PolicyTypeConfig code="prop_comm"/>
  <PolicyTypeConfig code="homeowners"/>
  <PolicyTypeConfig code="gen_liability"/>
  <PolicyTypeConfig code="inlandmarine"/>
  <PolicyField fieldName="Account"/>
  <PolicyField fieldName="EffectiveDate"/>
```

```
<PolicyField fieldName="ExpirationDate"/>
</PolicyPeriodDef>
```

This definition specifies that a single aggregate limit applies for *all* policies that:

- are of type commercial property, homeowners, general liability, or inland marine *and*
- have the same Account value *and*
- have the same effective date *and*
- have the same expiration date

Aggregate Limit Used Recalculation

Any time that you make relevant changes to claim data, ClaimCenter automatically recalculates how much of an aggregate limit the change used. As a general rule, you do not need to manually recalculate the used portion of your limits. However, if it becomes necessary for you to perform a manual recalculation, ClaimCenter makes it possible for you to do so.

If any of the database consistency checks that ClaimCenter performs on database tables fail, then you need to instruct ClaimCenter to recalculate the aggregate limit values. You also need to instruct ClaimCenter to repopulate the database tables. There are two commands you use: one to recalculate all limit used values and another to recalculate only those values that show as invalid.

To force a recalculation of all limit used values stored in the denormalized table, execute the following from the command line:

```
maintenance_tools.bat -password pwd -startprocess aggregateLimitCalculations
```

Note: You need run this process only if you encounter consistency check failures, and only if you cannot identify the reason for the inconsistency. You must run this command manually. You cannot schedule it to run periodically.

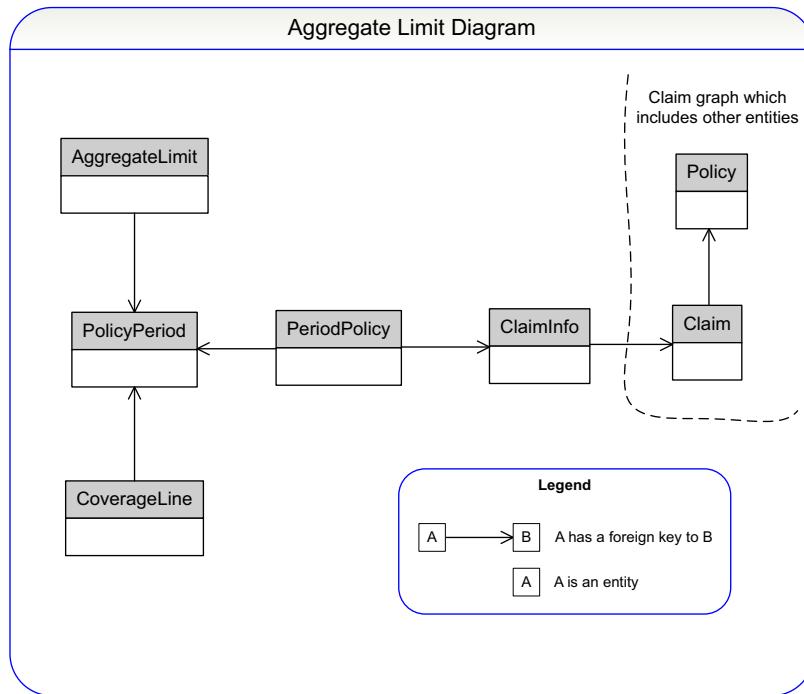
Guidewire also recommends that you always rebuild the aggregate limit calculations after you import financial and limit information. To rebuild, use the following command:

```
maintenance_tools.bat -password pwd -startprocess aggregateLimitLoaderCalculations
```

For information on using aggregate limits in Gosu business rules, see the *ClaimCenter Rules Guide*.

Aggregate Limit Diagram

The following graphic depicts a simplified view of how Aggregate Limits are 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 Plugin

During the policy retrieval process, the **Claim** has not yet been associated to 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.

Consequences

- You cannot add aggregate limits inside the policy plugin.
- You can create them later in Claim Pre-update rules.
- Another option is to add them to the claim from the user interface.

Specifying the Subtabs on a Policy

You can customize the subtabs that appear on the **Policy** page of a claim file. For example, an automobile policy can include a subtab with a list of vehicles and a property policy can include a subtab with a list of locations.

The **Policy** page always contains the **General** subtab, and you can add one or more other subtabs. The **PolicyTab.xml** typelist defines available subtabs that you can add.

You specify the subtabs in the **PolicyType.xml** typelist, which you access through the ClaimCenter **Lines of Business** editor. To define a policy type, use the **Categories** subtab to add a typelist (and typecode) from the **PolicyTab** typelist.

In addition, if the subtab you want to add is an **Aggregate Limits** subtab, you must add the policy type to the two following configuration files:

- **aggregateLimitUsed-config.xml** – For more information on this configuration file, see “Define What Counts Against an Aggregate Limit” on page 592.
- **policyPeriod-config.xml** – For more information on this configuration file, see “Configuring Policy Periods” on page 593.

Defining Internal (ClaimCenter-only) Policy Fields

Typically, applications external to ClaimCenter store the primary policy data. Thus, ClaimCenter imports policy snapshots for use with claims.

- 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.

For example, you might configure validation rules so that you can only make payments on a claim that has verified policies associated with it.

It is often useful, though, to have a hybrid policy structure. This means that you can maintain a connection to an external application with a verified policy, but also add additional policy data fields for use in ClaimCenter. To do this, you add *internal-only* fields to a policy. ClaimCenter manages these fields 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-only fields without causing the policy to be deverified.

To create internal fields

While the base configuration does not contain internal-only fields, you can create them by defining internal-only fields on the **Policy** entity (**Policy.etc**) and modifying the targeted .pcf files in Studio. The **ClaimPolicyGeneral.pcf** file contains multiple **Edit** buttons and each one contains the following behaviors.

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 internal-only fields if the policy has internal-only fields and the user has the permissions to edit them

This example explains how to add a custom column on an auto policy.

1. One way to create the **Policy.etc** file is to find the **Policy.eti** file, right click your mouse on the file, and select **Create extension file**.

Internal-only fields can be regular columns, foreign keys, or array properties. Use the **<internalonlyfields>** element to list the fields that are internal to ClaimCenter, and then the **<internalfield>** element to identify a particular field. To designate multiple fields as internal-only, list multiple **<internalfield>** elements, with each specifying one field.

2. 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. In this example, open the `PolicyGeneralPanelSet.Auto.pcf` file and add a new field that is internal-only. One way is to find the Type input and duplicate it. Ensure in the **Basic properties** tab that it is editable, giving it 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

1. Test your work by opening an auto claim in ClaimCenter and navigating to the **Policy** link on the left pane. Selecting it shows the **Policy:General** screen.
2. If you select **Edit**, notice you can either modify the entire policy or ClaimCenter-only fields. Select the latter.
3. Edit the field and save your changes. Notice that under the **Other** section, the **Verified Policy** field is still set to **Yes**.

Configuring Financials

This topic explains the data model and financial values managed by the ClaimCenter Financials feature. It explains how to configure both the behavior and the ClaimCenter interface to better match your business practices.

This topic includes:

- “Understanding Financials” on page 599
- “Financial Summary Screen Configuration” on page 605
- “Configuring Reserve Behavior” on page 610
- “Checks and Payments Configuration” on page 615
- “Bulk Invoice Payment Configuration” on page 619

See Also

- “Configuring Currency” on page 623
- “Working with Money and Currency Data Types” on page 304
- “Claim Financials” on page 131 in the *Application Guide*
- “Multiple Currencies” on page 171 in the *Application Guide*

Understanding Financials

This topic provides an overview of the concepts and data model underlying the Financials feature. It also provides an overview of the key configuration points within ClaimCenter for financial data. For more information, also see the following:

Topic	See
Overview of the financials system in ClaimCenter	“Claim Financials” on page 131 in the <i>Application Guide</i>
<i>Integration with external systems</i>	“Financials Integration” on page 195 in the <i>Integration Guide</i>

Topic	See
<i>Checks and payments</i>	"Checks" on page 143 in the <i>Application Guide</i> and "Payments" on page 139 in the <i>Application Guide</i>
<i>Financial batch processes</i>	"Batch Processes and Distributed Work Queues" on page 134 in the <i>System Administration Guide</i>
<i>Scheduling batch processes</i>	"Scheduling Batch Processes and Distributed Work Queues" on page 143 in the <i>System Administration Guide</i>
<i>Financial calculations</i>	"Financial Calculations" on page 167 in the <i>Rules Guide</i>
<i>Gosu rules relating to financials</i>	<ul style="list-style-type: none"> • "Approval Routing" on page 46 in the <i>Rules Guide</i> • "BulkInvoice Approval" on page 52 in the <i>Rules Guide</i> • "Initial Reserve" on page 62 in the <i>Rules Guide</i> • "Transaction Approval" on page 78 in the <i>Rules Guide</i>

Double-entry Accounting

ClaimCenter financial accounting uses the standard accounting principles of double-entry accounting or *T-accounts*. T-accounts serve these purposes:

- To record a history of changes to transactions
- To maintain running totals that support calculation of values across *groups* of transactions

A *group* is typically a *reserve line*. ClaimCenter organizes T-accounts by reserve line. A reserve line groups the financial activity for one combination of claim, exposure, cost type, and cost category. In fact there is a T-account type for every reserve line. Examples of T-account types are *cash out*, *draft payments*, *pending cash out*, and so forth. So, if a payment's status changes in a way that represents a cash outflow, then a line item is recorded in the *cash out* T-account for that payment's reserve line. The T-account's running balance is then adjusted accordingly. The line items provide for an increased ability to audit ClaimCenter data. The running balance provides an easy way for ClaimCenter to calculate, for example, the cash outflow for a reserve line without having to step through the transactions.

As you work within ClaimCenter, you *do not work with these accounts directly* as ClaimCenter maintains these accounts internally. Instead, you interact with objects in the ClaimCenter interface that behave in accordance with the double-entry principles: payments, recoveries, reserves, and so forth.

The ClaimCenter data model does contain T-account objects. These objects appear in the *Data Dictionary* and include the following:

TAccount	Represents one T-account, which maintains the balance of transactions contributing to it. Which transactions contribute to it depends on the T-account's type on one reserve line, along with a history of changes to that balance.
TAccountLineItem	Represents a change to a T-account's balance, created when a transaction's status changes.
TAccountTransaction	Effectively links a TAccountLineItem to the Transaction entity for which it was created.
Taccttxnhistory	This is an edge foreign key table to record the sequence of TAccountTransactions for one reserve line.

IMPORTANT ClaimCenter overwrites the T-account entities during staging table load, but only for the transactions being imported. Existing transactions are not affected. These entities are final.

Overview of the Financials Data Model

This section describes the financial data model with which users interact in the ClaimCenter interface and that you encounter as you configure financials. The ClaimCenter *Data Dictionary* contains many financial related

entities. The following list describes the key financial entity objects that you need to know to manage your configuration:

Entity	Description
Payment	A subtype of Transaction representing money paid out. A payment can be <i>eroding</i> or <i>non-eroding</i> depending on whether it draws down reserves for its reserve line.
Recovery	Records money received from such sources as subrogation, salvage, and so forth. This is a subtype of Transaction.
RecoveryReserve	Records an expected recovery. This is a subtype of Transaction.
Reserve	Records a potential liability. This is a subtype of Transaction. A Reserve is any transaction that designates money for payments. Typically, you set a reserve soon after a claim is made.
ReserveLine	A unique combination of Claim, Exposure, CostType, and CostCategory against which you can create reserves and apply payments or recoveries. Reserves are sometimes used to describe the amounts of reserves minus the amounts of payments on a reserve line or claim (this is a simplified description).
Transaction	Represents a financial transaction for a particular claim or exposure. This abstract class is a supertype. Most users manipulate and interact with its subtypes: Payment, Reserve, Recovery, and RecoveryReserve. Also, CostType and CostCategory are non-nullable properties, so every Transaction has a CostType and CostCategory.
TransactionLineItem	Used to further categorize a transaction. Each transaction has one or more line items.

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. The default cost types are:

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.

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 LineCategories available in your installation to reflect your business practices.

Transaction States and the Financial Building Blocks

You can think of transactions 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 the type key to TransactionStatus. Review the TransactionStatus typelist in the *Data Dictionary* for a full list of the possible transaction statuses.

The following list describes the key status values and their meanings:

Status	Meaning
awaitingsubmission	The transaction is waiting to be sent to the accounting subsystem on ScheduledSendDate (exists on the Check entity). This status applies only to payments and offsetting reserves.
pendingapproval	The transaction is waiting for approval by a supervisor.
submitted	ClaimCenter submitted the transaction to the accounting subsystem.

Running Totals

For each claim in the application, ClaimCenter maintains the *running total* of the financial transactions for a claim, either submitted, awaiting submission, or pending approval. ClaimCenter categorizes these running totals by transaction type (Reserve, Payment, and Recovery) and transaction status. T-accounts maintain the running totals for each reserve line. These T-accounts contain financial expressions and financial calculations.

- A *financials expression* is the sum (or difference) of T-account balances, and expressions themselves can be addition and subtraction.
- An expression creates a *financials calculation*. An example of this is ClaimCenter calling the `gw.api.financials.FinancialsCalculationUtil.getFinancialsCalculation` method.

A calculation can be applied to the T-accounts for a claim, exposure, or reserve line. This is accomplished by calling the `FinancialsCalculation.getAmount` method. It selects the T-accounts and adds their values according to the underlying expression and returns the result as a `CurrencyAmount`. In the base application configuration, Guidewire provides a set of pre-defined financial expressions each of which returns a different *financial calculation*.

In the base configuration, these financial calculations are used to compute the values seen on the **Financial Summary** page using methods on the `gw.api.financials` package. For example, you can access the Available Reserves financials expression value using the following Gosu:

```
gw.api.financials.FinancialsCalculationUtil.getAvailableReservesExpression()
```

You can also add and subtract the financial expressions from each other to arrive at your own custom values:

```
gw.api.financials.FinancialsCalculationUtil.getFuturePaymentsExpression().  
plus(gw.api.financials.FinancialsCalculationUtil.getTotalPaymentsExpression())
```

The following table lists the major building block APIs:

Building Block API	Definition
getAvailableReservesExpression	Total Reserves - (approved payments + Pending Approval Eroding Payments) AVAILABLE_RESERVES can be a negative value.
getFuturePaymentsExpression	All approved payments whose schedule send date is after today.
getGrossTotalIncurredExpression	OPEN_RESERVES + TOTAL_PAYMENTS
getOpenRecoveryReservesExpression	Sum of all submitted recovery reserves minus the sum of all submitted recoveries on the claim.
	By default, ClaimCenter automatically approves all recovery reserves. So, this value typically includes all recovery reserves. You can change this default behavior using an approval rule. See the <i>ClaimCenter Rules Guide</i> for more information.
getOpenReservesExpression	TOTAL_RESERVES - Total non-eroding payments
getPendingApprovalErodingPaymentsExpression	Sum of all payments that erode reserves and for which approval is pending.
getPendingApprovalNonErodingPaymentsExpression	Sum of all payments that do not erode reserves and for which approval is pending.
getPendingApprovalPaymentsExpression	PENDING_APPROVAL_ERODING_PAYMENTS + PENDING_APPROVAL_NONERODING_PAYMENTS
getPendingApprovalReservesExpression	Sum of all reserves pending approval.
getRemainingReservesExpression	This value is similar to open reserves except that it includes future eroding payments. It is the TOTAL_RESERVES - minus all approved eroding payments. This calculation can have a negative value.
getTotalIncurredNetRecoveriesExpression	REMAINING_RESERVES + FUTURE_PAYMENTS + TOTAL_PAYMENTS - TOTAL_RECOVERIES
getTotalIncurredNetRecoveryReservesExpression	REMAINING_RESERVES + eroding FUTURE_PAYMENTS + TOTAL_PAYMENTS - TOTAL_RESERVES
getTotalPaymentsExpression	Sum of all eroding and non-eroding payments submitted to the accounting subsystem and the awaiting submission payments with a scheduled send date of today or earlier.
getTotalPaymentsWithPendingExpression	TOTAL_PAYMENTS + FUTURE_PAYMENTS + PENDING_APPROVAL_PAYMENTS
getTotalRecoveriesExpression	Sum of all submitted recoveries on the claim.
getTotalRecoveryReservesExpression	Sum of all recovery reserves.
getTotalReservesExpression	Sum of all reserves submitted and awaiting submission.
getTotalReservesWithPendingExpression	TOTAL_RESERVES + future reserves + PENDING_APPROVAL_RESERVES

Note: You can also manipulate these building blocks through Gosu to create your own financial calculations. For more information, see “Financial Calculations” on page 167 in the *Rules Guide*.

Files that Manage Financials

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 payment plugin.
CheckBatching	ClaimCenter populates the Check Instructions drop-down list on the check wizard from this list.
CheckHandlingInstructions	Specifies handling instructions for a check. ClaimCenter populates the Check Instructions 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 CostCategory typelist.
CoverageType	Types of coverage available on a claim. ClaimCenter uses the values in this typelist as a key filter for the CostCategory typelist.
DeductionType	Types of deductions that can appear on a check.
FinancialSearchField	Search fields used while searching for checks or recoveries.
LineCategory	Populates the Category drop-down on the Payment Information page of the check wizard.
PaymentMethod	Requested payment method for all payments in a check.

Note: ClaimCenter stores the PCF files for configuring the financial screens in PCF → claim → financials.

Configuration Parameters

The following tables lists configuration parameters (defined in `config.xml`) that you can use to globally configure financials in ClaimCenter.

Parameter	Description
AllowMultipleLineItems	Specifies whether a transaction can have more than one transaction line item. <ul style="list-style-type: none"> If true (the default), a transaction can have multiple transaction line items. If false, a transaction can have one and only one transaction line item.
AllowMultiplePayments	Specifies whether a check can have more than one payment. The default is true. If false, a check can have one and only one payment.
AllowPaymentsExceedReservesLimits	Specifies whether a payment can exceed the available reserves. The default is false—a user cannot create a payment that exceeds the available reserves for the payment's reserve line. The exceptions are supplemental payments, manual checks, and first and final payments. They can always exceed reserves. If true, users can submit payments that exceed available reserves. However, the <code>PaymentsExceedReserves</code> authority limit set for a user can limit this ability.
CalculateBackupWithholdingDeduction	Specifies whether ClaimCenter calculates backup withholding for checks (if this applies). The default is true. If false, ClaimCenter never calculates backup withholding.
CheckAuthorityLimits	Specifies whether ClaimCenter verifies authority limits automatically any time that a user submits a group of items (reserves or payments) for approval. The default is true. You can check authority limits manually through rules. For more information, see the "Transaction Approval" on page 78 in the <i>Rules Guide</i> .
CloseClaimAfterFinalPayment	Determines whether ClaimCenter closes a claim after a user makes a final payment that closes the last open exposure. The default is true.
CloseExposureAfterFinalPayment	Determines whether ClaimCenter closes an exposure after a user makes a final payment. The default is true.

Parameter	Description
DefaultApplicationCurrency	This defines the default currency for all Financials. Financials values are tracked in this currency. The default is for USD (US dollars.)
MulticurrencyDisplayMode	This is the currency display mode for claims. The default is SINGLE. To change the user interface so that you can use multiple currencies, change it to MULTIPLE.
PaymentLogThreshold	Specifies a threshold value. ClaimCenter records payments greater than this threshold in the ClaimCenter log.
UseDeductibleHandling	If this is set to true, then AllowMultipleLineItems must be set to true as well.
UseRecoveryReserves	Specifies whether recovery reserves are available in ClaimCenter. The default is true. If false, users cannot create recovery reserves and the Open Recovery Reserves field in ClaimCenter becomes meaningless. In that case, remove the field from the Financial Summary screen. Currently, you can import recovery reserves regardless of the setting of this parameter.

Financial Summary Screen Configuration

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
- Configuring the Drop-Down
- Defining the Model Used by a Panel Set
- Controlling the Display of the Financial Model

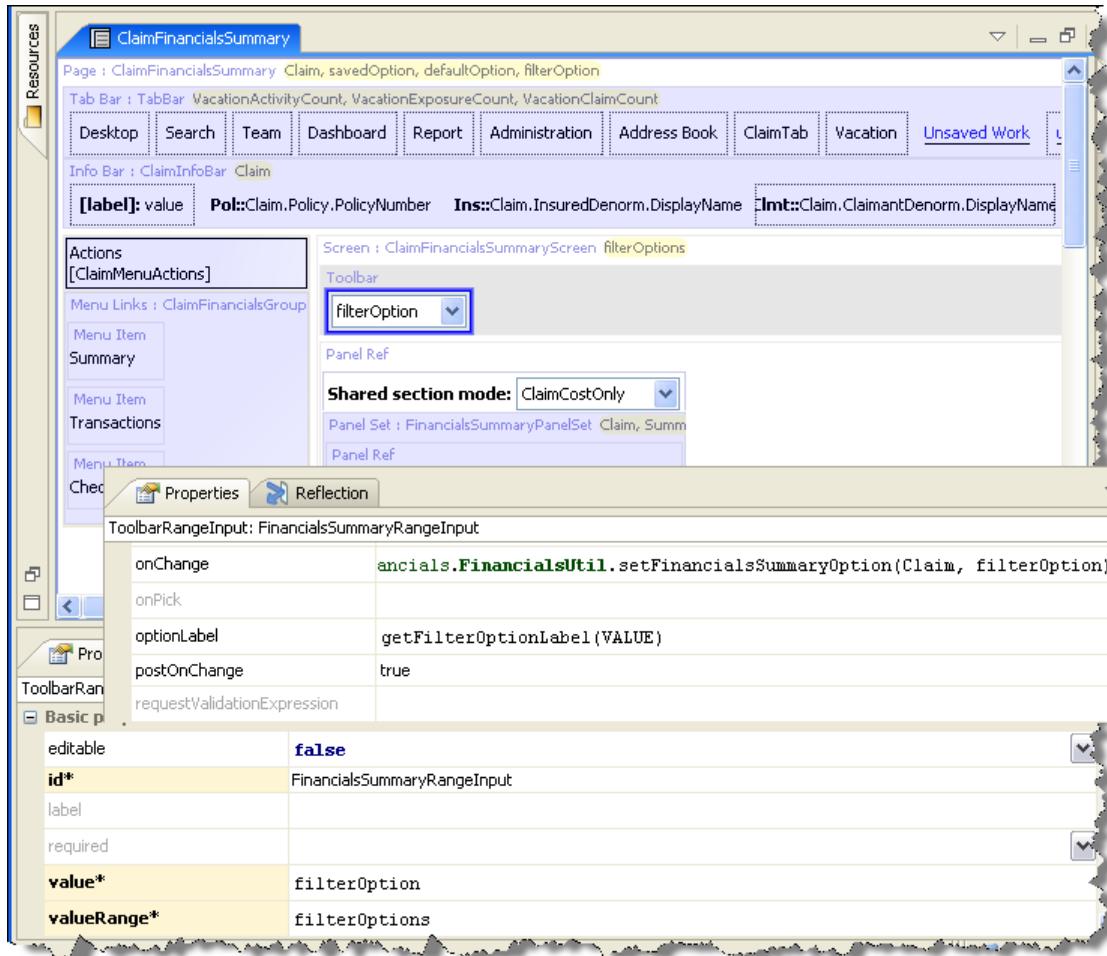
The Financials Summary Page

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 menu.
FinancialsSummaryLV	Defines the list view.
FinancialsSummaryPanelSet.Mode	Defines a model that corresponds to an item in the drop-down menu. ClaimCenter renders a separate panel (distinguished by <i>Mode</i>) for each item in the menu. Modes include: <ul style="list-style-type: none"> • Claimant • ClaimCostOnly • Coverage • Exposure • ExposureOnly

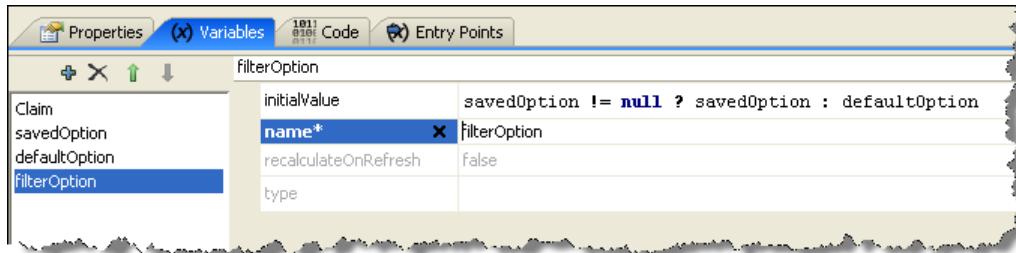
Configuring the Drop-Down

The `ClaimFinancialsSummary` file contains a `Toolbar` element that defines the option 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:

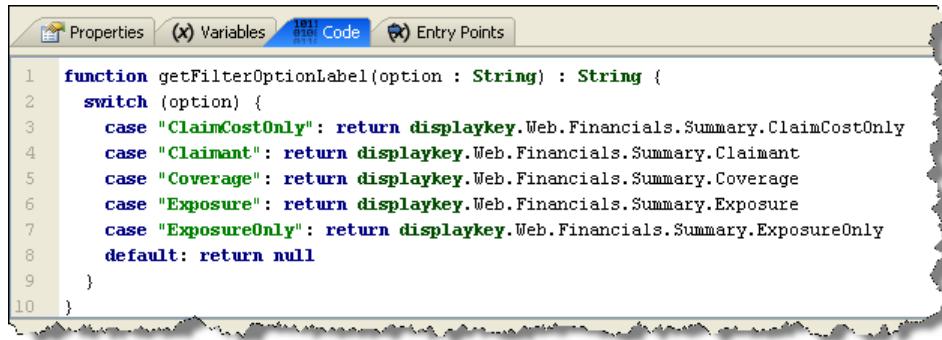
```
financials.FinancialsUtil.setFinancialsSummaryOption(Claim, filterOption)
```

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:

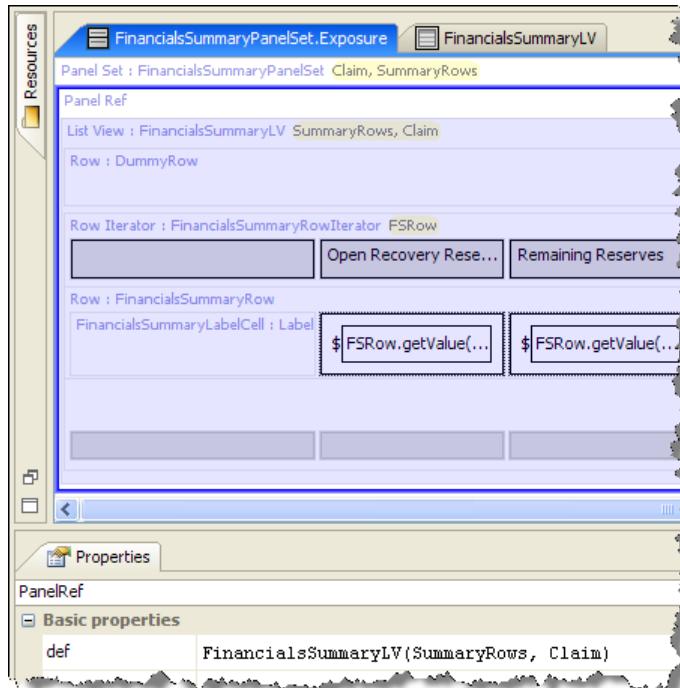
```
getFilterOptionLabel(VALUE)
```

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, there is a panel set file. For example, this is the Exposure mode of the `FinancialsSummaryPanelSet`. 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.FinancialsSummaryLevel.EXPOSURE,
    financials.FinancialsSummaryLevel.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:

- `arg1` Required. This is always set to `Claim`.
- `arg2` Required. The first level in an object summary.
- `arg3` Optional. The second level in the object summary.
- `arg4` A `FinancialExpression` array representing the column headings. You can specify the headings in any order.
- `arg5` A Boolean value indicating whether the panel is *claim cost* 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. (See “Transaction States and the Financial Building Blocks” on page 601.) 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:

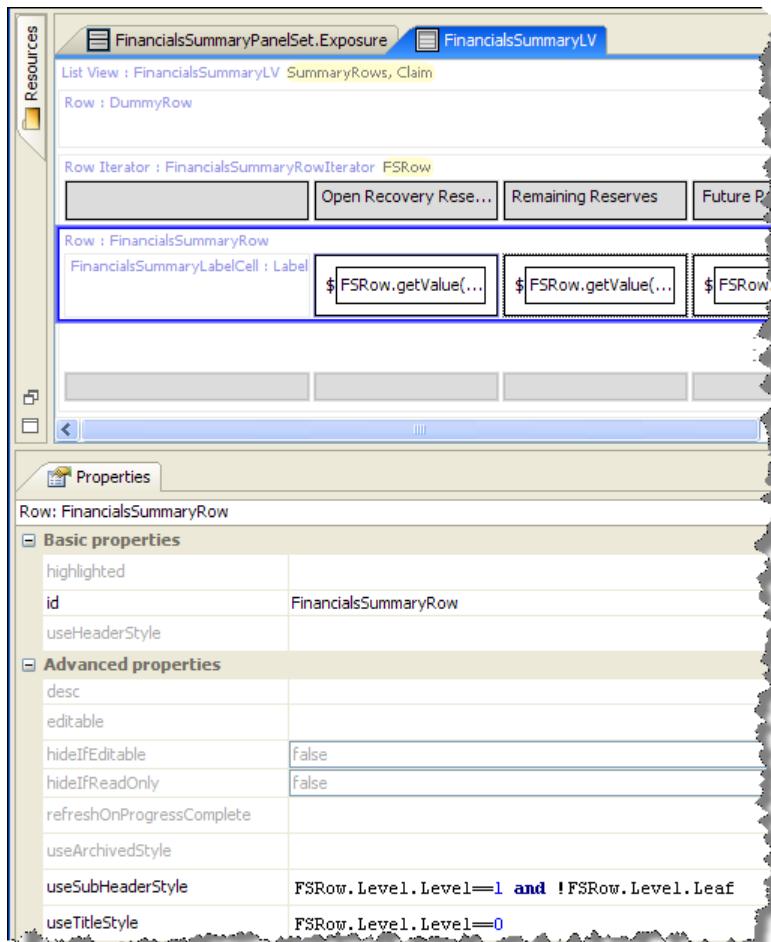
```
gw.api.financials.FinancialsCalculationUtil.getFuturePaymentsExpression()  
plus(gw.api.financials.FinancialsCalculationUtil.getTotalPaymentsExpression())
```

You can only use `.plus` or `.minus` to combine the financial building block values with the `FinancialsCalculationUtil` API.

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

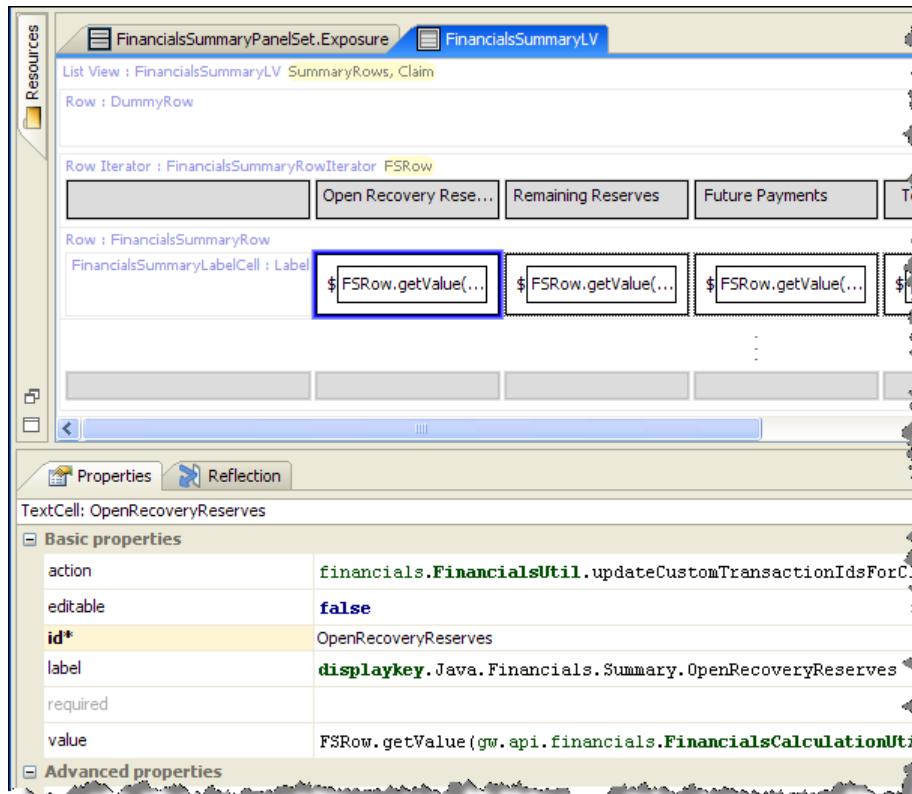
single `FinancialsSummaryRow` element containing multiple `FinancialSummaryCell` subelements. Each cell represents a column in the row.



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. 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 role is invisible.

Configuring Reserve Behavior

This section explains how to configure the behavior of the reserves and their related dialogs in ClaimCenter.

Understanding How Configuration Impacts Reserves

The **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*.

Set reserves by available reserves

It is possible to change the state of the reserves on a claim by changing the value of the **New Available Reserves** 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 **New Total Incurred** field for a reserve line.

To set the mode to one or the other, edit the **SetReservesByTotalIncurred** attribute 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.

Set Reserves

Save | Cancel | Add | Remove | Show Group | Show All | Link Document

All line items added or changed below will be submitted as a group. Any line item with no change will not be saved. Any line item with Pending Approval reserves that has its New Available Reserves set to equal its Currently Available reserves will have those Pending Approval reserves deleted. Comments are only saved if another field on the line has changed.

	Exposure	Coverage	*Cost Type	*Cost Category	Currently Available	Pending Approval	*New Available Reserves	Change	Comments
<input type="checkbox"/>	(1) 1st Party Vehicle - Ray Newton	Collision	Claim Cost	Auto body	\$400.00	-	\$ 400.00		
<input type="checkbox"/>	(1) 1st Party Vehicle - Ray Newton	Collision	Expense - A&O	Other	-	-	\$ 0		
<input type="checkbox"/>	(2) 1st Party Med Pay - Stan Newton	Medical payments	Claim Cost	Medical	\$2,000.00	-	\$ 2000.00		
<input type="checkbox"/>	(2) 1st Party Med Pay - Stan Newton	Medical payments	Expense - A&O	Other	-	-	\$ 0		
<input type="checkbox"/>	(3) 3rd Party Vehicle - Bo Simpson	Liability - Property damage	Claim Cost	Auto body	\$4,000.00	-	\$ 4000.00		
<input type="checkbox"/>	none (Claim)	<none	<none	<none			\$ 0		
Sum:					\$16,400.00	-	\$16,400.00		

Documents Linked to Reserves

Remove | Name | View | Type | Status | Author | Date Modified

Each line in this dialog 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** sets which PCF page defines the reserve line items:

- If **SetReservesByTotalIncurred** is **true**, the **EditableReservesLV.SetByNewTotalIncurred** file defines the content for the page.
- If it is **false**, **EditableReservesLV.SetByNewAvailableReserves** defines the page.

Note: 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
Available	The calculated amount of available reserves for this ReserveLine. This is a read-only field.
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"> • If SetReservesByTotalIncurred is false, the field contains difference between the New Available Reserves and Available Reserves columns. • If SetReservesByTotalIncurred is true, the field contains the difference between the New Total Incurred and Total Incurred columns.
Cost Category	The cost category corresponding to the reserve. This is a read-only field for an existing ReserveLine.
Cost Type	The cost type corresponding to the reserve. This is a read-only field for an existing ReserveLine.
Coverage	The policy coverage corresponding to the reserve. This is a read-only field.
Exposure	The claim's exposure corresponding to the reserve line. This is a read-only field for an existing ReserveLine.
New Available Reserves	If SetReservesByTotalIncurred is true, this field does not appear on the page. If SetReservesByTotalIncurred is false, users 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"> • If ClaimCenter displays the Set Reserve page, this field is equal to Available Reserves plus the Pending Reserves. • If a user enters a value in this field, the value represents the reserve the user wants for the ReserveLine. • After a user clicks Save, if the user requires approval for a reserve, ClaimCenter updates the Pending Approval field. Otherwise, it updates Available Reserves.
New Total Incurred	If SetReservesByTotalIncurred is false, this field does not appear on the page. If SetReservesByTotalIncurred is true, 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"> • If ClaimCenter displays the Set Reserve page, this field is equal to Total Incurred plus the Pending Reserves. • If a user enters a value in this field, the value represents the reserve the user wants for the ReserveLine. • After a user clicks Save, if the user requires approval for a reserve, ClaimCenter updates the Pending Approval field. Otherwise, it updates Total Incurred.
Pending Approval	Pending approval reserves for this reserve line. This field is read-only.
Total Incurred	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										
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				
Sum:				\$400.00	-	\$400.00				

After you enter a new reserve value, ClaimCenter reflects the change:

Filtered by: A Reserve										
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	-	\$700.00	\$300.00			
Sum:				\$400.00	-	\$700.00	Get Sum			

Then, suppose that you save the change, exit the dialog, and come back to it later. You see:

Filtered by: A Reserve										
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	\$300.00	\$700.00				
Sum:				\$400.00	\$300.00	\$700.00				

Suppose that you continue to change the reserve even more:

Filtered by: A Reserve										
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	\$300.00	\$900.00	\$500.00			
Sum:				\$400.00	\$300.00	\$900.00	Get Sum			

The next time you return to the dialog, if ClaimCenter has not yet approved the changes, you see the following:

Filtered by: A Reserve										
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	\$500.00	\$900.00				
Sum:				\$400.00	\$500.00	\$900.00				

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 single reserve change

Note: 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 the appropriate PCF file for the Set Reserve dialog.
 - If `SetReservesByTotalIncurred` is true, the `EditableReservesLV.SetByNewTotalIncurred` file defines the page content.

- If it is `false`, `EditableReservesLV.SetByNewAvailableReserves` defines the page.

Note: You can find these 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.
5. Rebuild and redeploy to the production ClaimCenter server.

Checks and Payments Configuration

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`, `final`, or `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 `partial` or `final` payment as non-eroding. Otherwise, these types always erode reserves. ClaimCenter maintains a running total of these payments in the following building blocks:

- `PENDING_APPROVAL_ERODING_PAYMENTS`
- `PENDING_APPROVAL_NONERODING_PAYMENTS`

You can reference these building blocks as you create custom calculated values.

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 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 35 for more information on configuration parameters.

Batch Processes for 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:

- *Checks and payments*—“Checks” on page 143 in the *Application Guide* and “Payments” on page 139 in the *Application Guide*
- *Financial batch processes*—“Batch Processes and Distributed Work Queues” on page 134 in the *System Administration Guide*
- *Scheduling batch processes*—“Scheduling Batch Processes and Distributed Work Queues” on page 143 in the *System Administration Guide*
- *Integrating with a check writing system*—“Financials Integration” on page 195 in the *Integration Guide*.

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:

- It updates all the associated T-accounts, unless the `taccountesc` process has already run and performed this task.
- It 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).
- It 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.
- It changes the `TransactionStatus` on all the associated payments to `submitting`.
- If a payment being escalated is final and has an exposure, its exposure is closed, provided it has no payments scheduled for sending after today.
- If a payment being escalated is final and is claim-level, its claim is closed provided it has no open exposures and no payments scheduled for sending after today.
- It runs the Transaction Post-Setup rules. If any of these rules 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.
- If the check is a recurring check and it is the second-to-last check to be submitted in the recurrence, then the system 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 12:01 a.m. and the second time at 5:01 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 T-Accounts Escalation Process

If a payment exceeds the available reserves, then it requires a zeroing-offset reserve to keep the available reserves of the reserve line on the payment from becoming negative. However, ClaimCenter does not create this zeroing-offset reserve until the payment check reaches its scheduled send date and the `taccountesc` batch process runs. This batch process does the following:

- Correctly updates the relevant T-accounts to reflect the fact that the check reached its send date.
- Creates any necessary zeroing-offsets for the payments for each check (if the payment exceeds reserves or is final). 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.

This 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. This gives the carrier time on the issue date of a check to make adjustments, while keeping summary financial values correct.

By default, `taccountesc` runs every 30 minutes between midnight and 4:30 a.m. 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 12:15 AM and 5:15 PM. You can reschedule the `bulkinvoiceesc` batch process by editing the `scheduler-config.xml` file. You can always run the process immediately on the command line:

```
maintenance_tools -startprocess bulkinvoiceesc -password <password> -username <username>
```

WARNING Do not schedule the `bulkinvoiceesc` process to run at the same time as the `financialsesc` process. Running these two at the same time can cause database concurrency issues.

The PendEscalationForBulk Property

Checks associated with a bulk invoice are *only* eligible for escalation if the `PendEscalationForBulk` property for a check is `false`. If this parameter is `true`, batch process `bulkinvoiceesc` escalates the check. You can use this parameter to escalate some bulk invoice checks while preventing others from being processed. You can use this, for example, to bundle newly arrived checks for the same vendor together.

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 is separate from what happens as ClaimCenter actually escalates the Check, which is determined by actions taken in Event Messaging (EM) rules and your messaging plugin implementations. See the *ClaimCenter Integration Guide* for more details about check and bulk invoice integration.

Scheduling the Financials Batch Processes

In the default application, the `financialsesc` batch process runs twice a day, at 12:01 a.m. and 5:00 p.m. If the `taccountesc` batch process does *not* run earlier in the day, the T-account entries and summary financial transactions are incorrect from midnight until noon. Depending on your implementation, you can schedule these two batch processes differently:

- Schedule one of these two processes to run before the calculated values need to be up to date.
- To keep checks editable until sometime during the last available working day, run `taccountesc` as soon after midnight as possible and schedule `financialsesc` at your midday time.
- If you do not care about not being able to edit future-dated checks that have reached their send date before `financialsesc` runs, then schedule `financialsesc` just after midnight. You need not run `taccountesc`.

Customizing the Check Wizard Recurrence Settings

Suppose that you would like to pre-populate some or all of the recurrence settings in the Check wizard with some of the values entered elsewhere in ClaimCenter. The recurrence settings appear in the [Entering Check Instructions](#) step of the Check wizard.

To pre-populate these fields, open the `PCF → claim → newtransaction → check → NewPaymentInstructionsDV` file and locate the `PaymentRecurrenceInput` widget.(It is the last widget on the page.) The `PaymentRecurrenceInput` widget has separate attributes that you can use to specify the default values of all the recurrence-related fields. These attributes are:

- `advanceDaysDefault`
- `inAdvanceDefault`
- `monthAbsCountDefault`
- `monthAbsDayDefault`
- `monthAbsDefault`
- `monthRelCountDefault`
- `monthRelDayDefault`
- `monthRelWeekDefault`
- `nonRecDateDefault`
- `numChecksDefault`
- `recDateDefault`
- `recurringDefault`
- `weekCountDefault`
- `weekDayDefault`
- `weeklyDefault`

You use a Gosu expression to set these values. For more information about the `PaymentRecurrenceInput` widget, see the [PCF Reference Guide](#). You can access the PCF reference from the Studio Help menu.

Customizing the Check Wizard's Default Payment Type

For a new payment on an open exposure, the `Payment Type` drop-down always has the following options:

- Partial
- Final

It is possible to default this control to one value or the other based on the selected `Cost Type` or some other value. You do this by editing the `PCF → claim → newtransaction → NewPaymentDetailDV` file. First, create a Gosu method that sets the `Payment.PaymentType` attribute. You can create this method either within the PCF itself or within a Gosu class. If you define the method in a Gosu class, you can call it in the `onChange` attribute of the `ReserveLine` selector control on `NewPaymentDetailDV`.

Bulk Invoice Payment Configuration

This section provides information about the support ClaimCenter provides for paying bulk invoices and contains the following:

- Overview of Bulk Invoices
- The Bulk Invoices Data Model
- Permissions for Bulk Invoice Processing

Overview of Bulk Invoices

The Bulk Invoices feature allows users 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 221 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 221 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” on page 52 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. This means that bulk invoice 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.

Entity	Description
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
BIValidationAlertType	Defines possible alerts returned from the validation plugin. This list contains a single alert type: • unspecified 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:

bulkinvcreate	Create a bulk invoice
bulkinvdelete	Delete a bulk invoice
bulkinvedit	Edit a bulk invoice
bulkinvview	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 622 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:

BulkInvoiceApprovalPattern	The name of the activity pattern to use while creating bulk invoice approval activities. By default, the name of the activity is approve_bulkinvoice.
AllowPaymentsExceedReservesLimits	This parameter applies to all payments, not just bulk payments. If this value is true, bulk payments can exceed reserves for each BulkInvoiceItem 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.isForBalkedCheck` 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.

Guidewire provides a sample rule to the Transaction Approval rule set that demonstrates the proper combined use of these methods. For more information about using these methods, see “[Transaction Approval](#)” on page 78 in the *Rules Guide*.

Configuring Currency

This topic describes how to configure Guidewire ClaimCenter to work with different currencies.

This topic includes:

- “Setting the Default Application Currency” on page 623
- “Setting a Currency Mode” on page 624
- “Working with Currency Typecodes” on page 625
- “Implementing a Single Currency Configuration” on page 625
- “Implementing a Multicurrency Configuration” on page 626
- “Changing the Default Application Currency” on page 626

Setting the Default Application Currency

You must set a default currency for Guidewire ClaimCenter. The default currency is the primary or operating currency in which you do business, regardless of whether you implement a single or multicurrency configuration.

You use configuration parameter `DefaultApplicationCurrency` in `config.xml` to set the default currency for the application as a whole. The value you choose for this configuration parameter must be a currency value that is a typekey value from the `Currency` typelist.

WARNING You **cannot** change the default application currency after you set it *without dropping the database*. If you attempt to do so, you can damage your installation. See “Changing the Default Application Currency” on page 626 for more information.

Setting a Currency Mode

You must set a currency mode for Guidewire ClaimCenter. The currency mode determines how ClaimCenter stores money values and displays them in the interface.

You use configuration parameter `MultiCurrencyDisplayMode` in `config.xml` to set the currency mode for the application. The currency mode is either *single currency* or *multicurrency*.

Set configuration parameter `MultiCurrencyDisplayMode` to one of the following values:

- SINGLE
- MULTIPLE

Note: It is possible to use the `money` data types and the `currencyamount` data types in both SINGLE and MULTIPLE currency modes. See “Working with Money and Currency Data Types” on page 304 for more information on the data types that you use with monetary values.

WARNING You **cannot** switch from one currency mode to the other (from SINGLE to MULTIPLE, or the reverse) after you set it *without dropping the database*. If you attempt to do so, you can damage your installation.

SINGLE Mode

In SINGLE currency mode, ClaimCenter assumes that all monetary amounts in the system are in the same currency. ClaimCenter uses the `<CurrencyFormat>` entries in each `<GWLocale>` in `localization.xml` to format the money amount, depending on the locale of each user. For example:

```
<Localization xmlns="http://guidewire.com/localization">
  <GWLocale code="en_US" name="English (US)" typecode="en_US">
    ...
    <CurrencyFormat negativePattern="($#)" positivePattern="$#" zeroValue="-" />
  </GWLocale>
</Localization>
```

As all money values are in the default currency, you must ensure that the `<CurrencyFormat>` for each `<GWLocale>` specifies the money format for that one currency. It is possible to set slightly different formatting based on local custom, but all money formatting must be for the one default currency.

MULTIPLE Mode

In MULTIPLE currency mode, ClaimCenter obtains money formatting information from `currencies.xml`. For example:

```
<CurrencyType code="usd" desc="US Dollar" storageScale="2">
  <CurrencyFormat positivePattern="$#" negativePattern="($#)" zeroValue="-" />
</CurrencyType>
```

In MULTIPLE mode, ClaimCenter ignores any `CurrencyFormat` information in file `localization.xml`. However, even though unused, a tag for the default currency must be present in file `localization.xml`, even in MULTIPLE mode.

See Also

- “`DefaultApplicationLocale`” on page 53
- “`DefaultApplicationCurrency`” on page 57
- “`MultiCurrencyDisplayMode`” on page 58
- “`Working with Locales`” on page 467

Working with Currency Typecodes

In the base configuration, ClaimCenter defines the following currencies in the **Currency** typelist:

- US Dollar
- Euro
- British Pound
- Canadian Dollar
- Australian Dollar
- Russian Ruble

The **Currency** typelist must include, at the very least, a single currency typecode that is the default application currency. You set the default application currency in file `config.xml`, through configuration parameter `DefaultApplicationCurrency`.

Mode	Action
SINGLE	If you implement SINGLE currency mode, then delete or retire all currency typecodes except for the default application typecode. Delete all unwanted currencies from file <code>currencies.xml</code> as well.
MULTIPLE	If you implement MULTIPLE currency mode, then delete or retire all currency typecodes except for those used for the currencies that you define in <code>currencies.xml</code> . If any of the requisite Currency typecodes do not exist, then you need to create them. Delete all of the currencies from file <code>currencies.xml</code> that you do not want and add the currencies that you do want.

Implementing a Single Currency Configuration

If you implement a single currency configuration, then do the following:

Location	Action
<code>config.xml</code>	<ul style="list-style-type: none">• Set configuration parameter <code>DefaultApplicationCurrency</code> to the operating currency.• Set configuration parameter <code>MultiCurrencyDisplayMode</code> to SINGLE. <p>WARNING You cannot change the <code>MultiCurrencyDisplayMode</code> parameter in a production environment without dropping the database.</p>
<code>currencies.xml</code>	Remove all currency entries except the entry for the default application currency.
<code>datatypes.xml</code>	Set the number of fractional or decimal digits to show for monetary values in the interface. The <code>precision</code> and <code>scale</code> attributes on <code><MoneyDataType></code> control the total number of digits to show and the number of digits to show after the decimal. These attributes control both how ClaimCenter stores monetary values in the database and how ClaimCenter displays a monetary value in the interface. It is possible that you need to display a different number of digits in the interface than what ClaimCenter stores in the database. If this is the case, then use the <code>appscale</code> attribute on <code><MoneyDataType></code> to control the display of digits in the interface. If you do not set an <code>appscale</code> value, then ClaimCenter uses the <code>scale</code> value. See the descriptions for the <code><MoneyDataType></code> attributes in “ Money Data Types ” on page 305. IMPORTANT The <code>appscale</code> attribute does not exist in the base configuration. You need to add this attribute to the <code><MoneyDataType></code> element if you intend to use it.
Currency typelist	<p>WARNING You cannot change the <code>scale</code> attribute without dropping the database.</p> <p>In the Studio Typelists editor, open the Currency typelist. Delete all currency typecodes except for the default currency typecode.</p>

Implementing a Multicurrency Configuration

If you implement a multicurrency configuration, then do the following:

Location	Action
config.xml	<ul style="list-style-type: none"> Set configuration parameter <code>DefaultApplicationCurrency</code> to the operating currency. Set configuration parameter <code>MultiCurrencyDisplayMode</code> to <code>MULTIPLE</code>. <p>WARNING You cannot change the <code>MultiCurrencyDisplayMode</code> parameter in a production environment without dropping the database.</p>
currencies.xml	<p>Remove all currency entries except those entries for the currencies in which you intend to make payments or otherwise store money amounts in the database. If entries for all the currencies in which you intend to do business do not exist, then add them.</p> <p>You set the number of decimal digits to show in the interface for each currency through the <code>storageScale</code> attribute on the individual <code><CurrencyType></code> elements for each currency.</p> <p>The <code>appscale</code> attribute on <code><MoneyDataType></code> in <code>datatypes.xml</code> is useful in multicurrency configurations for the money data types. However, with <code>currencyamount</code> data types, the <code>storageScale</code> parameter supersedes the <code>appscale</code> attribute. Make the currency <code>storageScale</code> value less than or equal to the <code>appscale</code> value and make the <code>appscale</code> value less than or equal to the <code>scale</code> value. The following must always be true.</p> $\text{storageScale} \leq \text{appscale} \leq \text{scale}$ <p>See the descriptions for the <code><MoneyDataType></code> attributes in “Money Data Types” on page 305.</p> <p>IMPORTANT The <code>appscale</code> attribute does not exist in the base configuration. You need to add this attribute to the <code><MoneyDataType></code> element if you intend to use it.</p>
localization.xml	In <code>MULTIPLE</code> mode, ClaimCenter ignores any <code><CurrencyFormat></code> information in <code>localization.xml</code> . However, even though unused, a tag for the default currency must be present in file <code>localization.xml</code> , even in <code>MULTIPLE</code> mode.
Typelists	<p>In the Studio Typelists editor, open the Currency typelist. Delete all currency typecodes except for those used for the currencies you defined in <code>currencies.xml</code>. If any of the requisite currency typecodes do not exist, then you need to create them.</p> <p>Note: If a typecode was in use in a previous version of the application, then retire the typecode instead.</p>

Changing the Default Application Currency

The base ClaimCenter configuration sets the default application currency to United States dollar. Changing the default application currency is a multistep process. The following steps illustrate this process of changing the default application currency from United States dollar to New Zealand dollar in Guidewire ClaimCenter.

WARNING You **cannot** change the default application currency after you set it *without dropping the database*. If you attempt to do so, you can damage your installation.

To change the default application currency

The following steps apply to both `SINGLE` and `MULTIPLE` currency mode.

1. Open `config.xml` and make the following changes:

```
<param name="DefaultApplicationLocale" value="en_NZ"/>
<param name="DefaultCountryCode" value="NZ"/>
<param name="DefaultApplicationCurrency" value="nzd"/>
```

2. Open the `LanguageType` typeplist and add `en_NZ` to the list of language types.

3. Open the `Currency` typeplist and add typecode `nzd` to the list of currencies.

- If a typeplist filter exists that uses `USD`, change the filter to use `NZD` instead.

- If using SINGLE mode, remove the other currency typecodes. This makes nzd the only entry.
4. Open the `EstDamageType` typelist. Select each code value in turn. For each one, change the `Code` value of `USD` to `NZD` in the `Category` filter at the bottom of the screen. Modify the dollar amounts as needed.
5. Open `localization.xml` and add a `<GWLocale>` locale for `en_NZ`. Set the attributes on `<CurrencyFormat>` correctly for the New Zealand currency.
- ```
<GWLocale code="en_NZ" name="English (NZ)" typecode="en_NZ">
 <DateFormat long="E, d MMMM yyyy" medium="d MMMM yyyy" short="d.M.yy"/>
 <TimeFormat long="H:mm:ss" medium="hh:mm:ss aa" short="hh:mm aa"/>
 <NumberFormat decimalSymbol="#" thousandsSymbol=","/>
 <CurrencyFormat negativePattern="($#)NZD" positivePattern="#NZD" zeroValue="-"/>
</GWLocale>
```
6. Open `currencies.xml` and add a `<CurrencyType>` element for `nzd`. Set the `<CurrencyFormat>` element attributes correctly for the New Zealand dollar. In `SINGLE` mode, remove all other `<CurrencyType>` elements.
7. For file `authority-limits.csv`, do the following:
- a. Copy file `<install>/modules/cc/config/import/gen/authority-limits.csv` to the following location:  
`<install>/modules/configuration/config/import/gen/`
  - b. Replace all occurrences of `usd` with `nzd`.
8. For file `system_data.xml`, do the following:
- a. Copy file `<install>/modules/xx/config/import/source/system_data.xml` to the following location:  
`<install>/modules/configuration/config/import/source/`
  - b. Replace all occurrences of `usd` with `nzd`.
9. Add the following files to `<install>/configuration/config/locale/en_NZ`:
- `display.properties`
  - `gosu.display.properties`
  - `studio.display.properties`
- Populate these files appropriately. See “Adding a New Locale” on page 468 for more information, especially “Step 4: Create and Populate the New Locale Folder” on page 473.
10. Add a `<Zones>` element to `zone-config.xml` and set the `countryCode` attribute to `NZ`. For example,
- ```
<?xml version="1.0" ?>
<ZoneConfig xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="../../xsd/zone-config.xsd">
  <Zones countryCode="NZ">
    <Zone code="postalcode" fileColumn="1" unique="true">
      <AddressZoneValue>Address.PostalCode</AddressZoneValue>
    </Zone>
  </Zones>
</ZoneConfig>
```
11. 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.
12. Drop the database, then restart the application server. This action loads the modified authority limit profiles.
13. Re-import the zone configuration data using the `zone_import` command. See “`zone_import` Command” on page 178 in the *System Administration Guide* for details.

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 629
- “Encrypting Claim Snapshot Fields” on page 630
- “Configuring Snapshot Templates” on page 631

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. An FNOL snapshot records the data available to a carrier at the time an insured first notifies the carrier of a loss. ClaimCenter creates an 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 an 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 an 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 an 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 and 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.** You must provide an algorithm.
- **The Encryption Upgrade work queue.** This work queue can be configured to run at a different time and 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

- “Batch Processes and Distributed Work Queues” on page 134 in the *System Administration Guide* to understand how the Encryption Upgrade work queue functions.
- “Encryption Integration” on page 401 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.

Note: 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 task 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 that contains an XML description of the claim's FNOL. Using this field has the following implications for data model extensions and changes:

- If you extend or change the ClaimCenter data model before the application captures an FNOL snapshot, ClaimCenter captures your extension in the snapshot if there is actual data to capture.
- Pre-existing snapshot data does not have any data model extensions or changes that you make subsequent to those snapshots.

Note: If a particular snapshot does not contain the data your page references, ClaimCenter displays the FNOL snapshot fields as empty.

Configuring Deductibles

This topic explains how deductibles are structured, and how they interact with checks.

This topic includes:

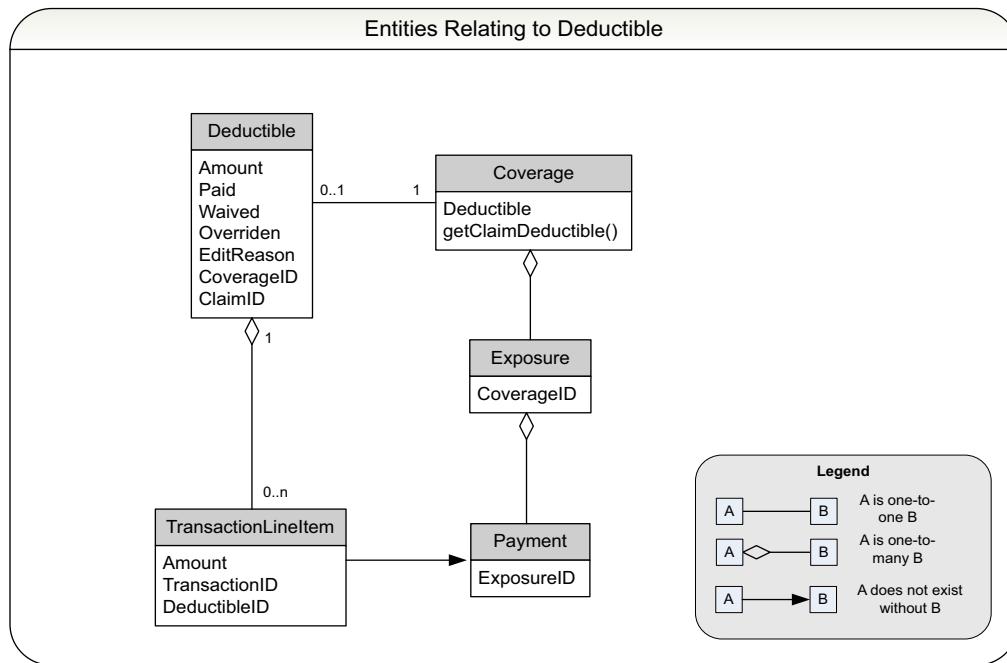
- “Deductible Data Model” on page 634
- “Typekeys” on page 635
- “Permissions” on page 635
- “Deductibles and Checks” on page 635
- “Deductibles and Rules” on page 637

See Also

- “Deductible Handling” on page 183 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**.

Typekeys

There are two typekeys that relate to deductibles, both for the Line Category 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 no longer. This is because it is on a recoded payment, or transferred or deleted check, or because it is on an onset payment whose deductible could not be applied to it.

Both typekeys are valid for any `Exposure/CostType/CostCategory` because a deductible can be applied to a payment of any reserve line. However, they are not valid for any matter. In any place in the user interface where `LineCategory` is editable, such as in Step 2 of the check wizard, the typekeys are filtered out from the list of available options. This filtering prevents the user from selecting either typekey for normal non-deductible line items. Anywhere in the user interface where the `LineCategory` shows, if a line item has a category of `Deductible` or `Former Deductible`, you cannot edit the line category and the amount.

Permissions

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 635
- “Recoding Payments” on page 636
- “Deleting and Voiding/Stopping Checks” on page 636
- “Denying or Resubmitting Checks” on page 636
- “Applying Deductibles on Multicurrency Checks” on page 636
- “Cleared or Issued Checks” on page 636
- “Cloning Checks” on page 636

Transferring Checks

When transferring a check, any deductible line items on the original and offset payments become `Former Deductible`, and linked deductibles are unlinked. These actions are performed through the method `check.unlinkDeductibles`. The method is called in the `doTransfer` method in `CheckTransfer.pcf` file after the target check has been created but before calling the `financials.CheckUtil.transferCheck` method.

For the onset payment, if the target exposure has a valid deductible, the target exposure's deductible is applied to the onset payment. If it has not been paid or waived, and the amount and the claim currency are equal, 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`. These actions are 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 that has a claim currency that is different from the original claim's currency, the deductible is not applied on the target claim. The deductible is not applied in this case 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. These actions are accomplished by calling the method `payment.unlinkDeductible` on the original payment. The method is called in the `doRecode` method in `RecodePayment.pcf` file after the onset payment has been created, but before calling `financials.FinancialsUtil.recodePayment`.

For the onset payment, if the target exposure has a valid deductible (see “Transferring Checks” on page 635), the target exposure's deductible is applied to the onset payment. Otherwise, no deductible is applied, and the onset payment's deductible line item instead become Former Deductible. These actions are 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, voiding, or stopping a check converts all its deductible line items to have the line category Former Deductible. This category is not exposed in the user interface in a deleted check. The linked deductibles are unlinked and unpaid. This action 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 tries to relink all the deductible line items to their prospective deductibles provided they are still valid. This means that they are not waived or paid, and they 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">• Update Deductible On Updated Exposure Coverage• Update Deductible On Updated Coverage Deductible
Transaction Pre-update	<ul style="list-style-type: none">• Unlink Deductible After Check Denial

See “Preupdate” on page 70 in the *Rules Guide* for more information on the Pre-update rules.

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 639
- “Catastrophes Data Model” on page 640
- “Catastrophe Configuration Parameter” on page 640

See Also

- “Catastrophes” on page 115 in the *Application Guide* to learn about catastrophes in general
- “Catastrophe Bulk Association” on page 118 in the *Application Guide* to learn about Catastrophe Bulk Associations batch process
- “Catastrophes” on page 402 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 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 641
- “Duplicate Claim Search” on page 642
- “Duplicate Check Search” on page 643

Understanding the Gosu Templates

You can modify the search criteria for duplicate claims and duplicate checks in the Gosu templates in Studio. Navigate to **Classes** → **gw** → **duplicatesearch**. The folder contains the following templates:

Gosu template	Description
<code>gw.duplicatesearch.DuplicateCheckSearchTemplate</code>	Duplicate Check search
<code>gw.duplicatesearch.DuplicateClaimSearchTemplate</code>	Duplicate Claim search

Parameters

The `DuplicateCheckSearchTemplate` takes three parameters:

- A `DuplicateSearchHelper`, which provides utilities for SQL construction.
- The Check to search for duplicates.
- A `checkBeingCloned` parameter. If the Check is a clone of an existing check, then 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 utilities 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 there are no matches, then the user does not see any messages.

The query considers the claim to be a duplicate if either

- The claim has the same policy and has a loss date 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 the template (`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 bolded text string (**Three**) to a different value (number), save your work, and restart the application server.

Duplicate Check Search

Gosu template `DuplicateCheckSearchTemplate` (used to create the SQL query for finding duplicate checks) verifies that the check has not already been created. If it finds any duplication, then a list of check IDs that match the check show in the user interface. 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. However, it is more complex than that. 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.

Configuring Claim Health Metrics

This topic explains how to configure Claim Health Metrics. You can add a new tier, a high-risk indicator, or a new claim metric. You could add a new tier for additional granularity. (Notice the exposure tiers are more granular than the claim tiers.) You could create a high risk indicator for anything that is important to your business. For example, you might want to 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 645
- “Adding a High-Risk Indicator” on page 647
- “Adding a New Claim Metric” on page 649

See Also

- “Claim Performance Monitoring” on page 315 in the *Application Guide* to learn about this feature.
- “Metrics and Thresholds” on page 418 in the *Application Guide* to learn how to administer claim health metrics.

Adding a New Tier

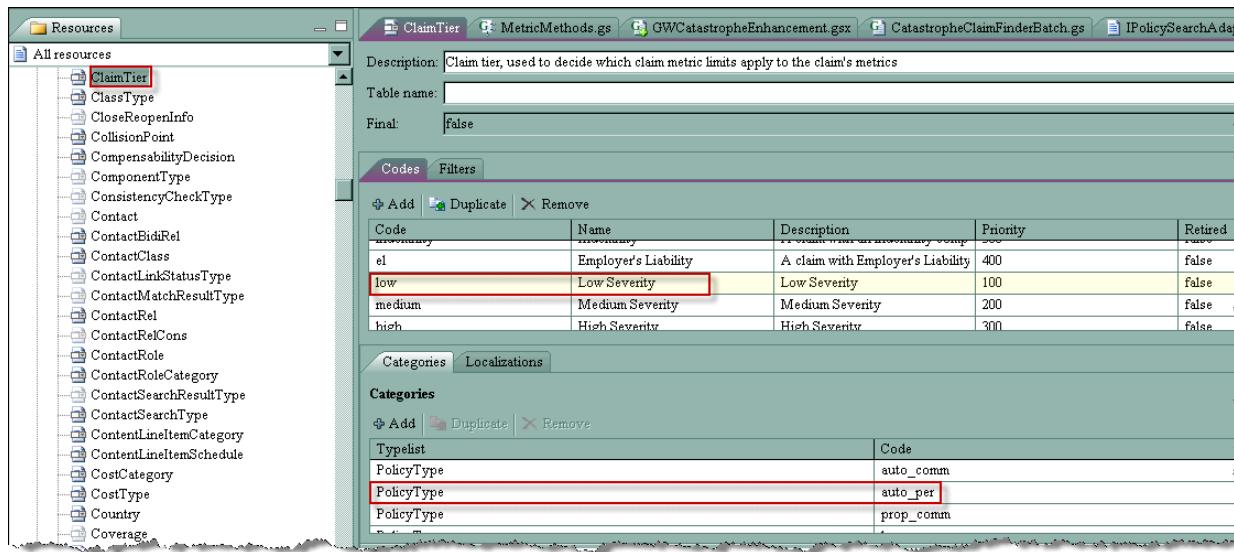
To add a new tier, create it first in Studio and define its associated limit values in the ClaimCenter Administration tab user interface, and then implement it with Gosu logic.

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 seen in the following example:



Each tier is associated to **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 the new target values in the ClaimCenter

1. Navigate to the **Administration** tab → **Metrics and Thresholds**.
2. Click **Edit** and select the new tier from the individual metric drop down menu.
3. Enter the target values for the new tier.
4. 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`
 - `FlagClaimIndicator`
 - `LargeLossClaimIndicator`
 - `LitigationClaimIndicator`
 - `SIUClaimIndicator`

2. Define a new entity subtype. The `supertype` you choose depends on the type of indicator that you want. For example:

```
<?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>
```

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 in 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" || 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`:

- a. Navigate to **configuration** → **Page Configuration (PCF)** → **claim** → **ClaimInfoBar**.
- b. 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**.
- c. 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.IsNullOr</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:

- a. Navigate to **configuration** → **Page Configuration (PCF)** → **claim** → **summary** → **indicator** → **ClaimIndicatorInputSet.LitigationClaimIndicator**.
- b. Right-click `ClaimIndicatorInputSet.LitigationClaimIndicator` and click **Duplicate**.
- c. 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**.
- d. 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.ExampleClaimMetricMethodsImpl"/>
    <implementsInterface
        iface="gw.api.claim.metric.RecalculateMetrics"
        impl="gw.claim.metric.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 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` type-list.

4. 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
<code>ExampleClaimMetric</code>	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 the claim gets new activities or current activities are changed or existing activities become overdue.

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.general

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 {
        return null //## todo: Implement me
    }
}
```

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.

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. Save your work in Studio

- 10.** If ClaimCenter is running, stop it. At the command prompt open to `ClaimCenter\bin`, press `CTRL+C` and then enter the following commands:

```
y  
gwcc dev-stop
```

- 11.** To ensure that your data model changes are correct, regenerate the data dictionary. At the command prompt, enter:

```
gwcc regen-dictionary
```

- 12.** Restart the server. At the command prompt, enter:

```
gwcc dev-start
```

- 13.** Log in to ClaimCenter as an administrator, such as user name `su` with password `gw`.

- 14.** Click the **Administration** tab and then click **Metrics and Thresholds** in the left info bar.

- 15.** Click **Edit** and, on the **Claim Metric Limits** tab, add limits for your new metric type. For more information, see “Metrics and Thresholds” on page 418 in the *Application Guide*.

- 16.** Click the **Administration** tab in the user interface, and then click **Metrics and Thresholds** in the left info bar.

- 17.** Click **Edit** and, on the **Claim Metric Limits** tab, add limits for your new metric type. For more information, see “Metrics and Thresholds” on page 418 in the *Application Guide*.

- 18.** Run batch processes as follows:

- a.** Press `ALT+SHIFT+T` to open the **Server Tools** tab.
- b.** Click **Batch Process Info** in the left info bar.
- 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 your implementation class implements the `RecalculateMetrics` interface, you can add the metric to claims with existing metrics by running the **Recalculate Claim Metrics** batch process.

Adding Your New Claim Metric or Indicator to Existing Claims

You can apply a new claim metric or indicator to claims that already have metrics. There are two ways to apply a claim metric, one of which is useful for indicators as well.

- Enable the **CER04000 Recalculate claim metrics** rule, which adds a new metric or indicator. See “Enabling the Claim Exception Rule” on page 652.
- If your implementation class implements the `RecalculateMetrics` interface, you can run the **Recalculate Claim Metrics** batch process. You can also set this batch process to run on a regular basis. For more information on batch processes, see “Batch Processes and Work Queues” on page 129 in the *System Administration Guide*.

Enabling the Claim Exception Rule

You can apply a new claim metric or indicator to claims that already have metrics by enabling the `ClaimException` rule **CER04000 Recalculate claim metrics**. This rule verifies that all claim metrics, exposure metrics, and claim indica-

tors are created on every claim. If there are any missing metrics or indicators in a claim, ClaimCenter adds them to the claim.

Guidewire disables this rule in the base configuration. To enable this rule, in ClaimCenter Studio, navigate to **configuration** → **Rule Sets** → **Exception** → **ClaimExceptionRules**, right click **CER04000 Recalculate claim metrics**, and click **Active**. If you see a message asking if you want to edit the file, click **Yes**.

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 655

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/editing exposures based on incidents in the user interface.

Implicit Incidents

After creating a new exposure, a new incident is also created, and the exposure and incident remain bound together from that point. 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) appears like a normal exposure field.

There are two exposure screens that exist in the New Claim wizard, and in the main claim file. The PCF file for creating an exposure on the new exposure pages appears as the following:

```
<Variable name="Exposure" type="Exposure" initialValue="Claim.newExposureWithNoIncident(CoverageType, CoverageSubtype, Coverage)" convertedFrom="LocalValue"/>
<Variable name="Incident" type="Incident" initialValue="Exposure.initializeIncident()" convertedFrom="LocalValue"/>
```

The exposure is created without an incident and the Gosu class library method `Exposure.initializeIncident()` then sets it up. It uses the `Exposure.newIncident()` method to immediately create a brand new incident for all exposure types except vehicle and property damage.

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 user interface page you see a drop down of all suitable incidents. You can also create a brand new incident. For these exposure types, the `Exposure.initializeIncident()` method attempts to pre-fill the incident picker however, you can always change the pre-filled value.

- 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/property on the policy.

- If this fails, the incident is just chosen 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 (and possibly incident) type, you have to decide whether to use the implicit or explicit incident approach. 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.

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, then 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 associate it with the exposure. Since anyone who uses such 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.

Injured is a Contact Role

The **Injured** contact role is the injured party associated with an injury incident. Navigate to: **Claim** → **Loss Details** → **Injured** which displays a list of these contacts. Each member of this list points to an instance of an injury incident.

Incidents Data Model

Gosu and Incidents

At the domain level, Gosu can work with incident types and their properties. The following are some Gosu properties and methods 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 `<IncidentType>sOnly`. 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 which 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.<IncidentType>sOnly` arrays when dealing with Incidents, since you are usually only interested in Incidents of a particular type. If you use `Claim.Incidents`, you see Implicit Incidents (see the previous section) 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.<IncidentType>sOnly` 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

Archiving Claims

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:

- “Archive-related Documentation” on page 665
- “Archiving and the Domain Graph” on page 666
- “Claims Archiving in Guidewire ClaimCenter” on page 667
- “Archiving and Encryption” on page 668
- “Selecting Claims for Archive Eligibility” on page 668
- “Restoring Claims from the Command Line” on page 669
- “Monitoring Claim Archiving Activity” on page 669
- “Configuring Claims Archiving” on page 670
- “The Archive Plugin” on page 673

Archive-related Documentation

See the following documentation related to archiving for more details.

See also

- “Archiving” on page 87 in the *Application Guide* – information on archiving claims, searching for archived claims, and restoring archived claims.
- “Archive Parameters” on page 39 in the *Configuration Guide* – discussion on the configuration parameters used in claims archiving.
- “Archiving Claims” on page 665 in the *Configuration Guide* – information on configuring claims archiving, selecting claims for archiving, and archiving and the object (domain) graph.

- “Archiving Integration” on page 285 in the *Integration Guide* – describes the archiving integration flow, storage and retrieval integration, and the `IArchiveSource` plugin interface.
- “Archive” on page 48 in the *Rules Guide* – information on base configuration archive rules and their use in detecting archive events and managing the claims archive and restore process.
- “Logging Successfully Archived Claims” on page 37 in the *System Administration Guide*.
- “Purging Unwanted Claims” on page 58 in the *System Administration Guide*.
- “Archive Info” on page 160 in the *System Administration Guide*.
- “Upgrading Archived Entities” on page 62 in the *Upgrade Guide*.

IMPORTANT To increase performance, most customers find increased hardware more cost effective than archiving unless their volume exceeds one million claims or more. Guidewire strongly recommends that you contact Customer Support before implementing archiving to help your company with this analysis.

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. For example, 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.

In order 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...
RootInfo	<p>During the archiving of an instance of the domain graph, ClaimCenter leaves behind (in the main ClaimCenter database) a skeleton entity that provides the following:</p> <ul style="list-style-type: none">• Sufficient information to restore the data.• Sufficient information for a minimal search on archived data. <p>This skeleton entity—and only 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 cannot change which entity implements the <code>RootInfo</code> delegate.</p>
Extractable	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.) The use of this delegate ensures the creation of the <code>ArchivePartition</code> column that ClaimCenter uses during the archive process.
OverlapTable	Overlap tables are tables in which each 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 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.

Data model delegate objects. A Delegate is a reusable type that defines database columns, 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. 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 187.
- For a discussion of the domain graph, see “The Domain Graph” on page 279.
- For a discussion of delegate objects and how to work with them, see “Delegate Data Objects” on page 197.

The ‘ClaimInfo’ Entity

As ClaimCenter creates a claim, it also creates a `ClaimInfo` entity. The `ClaimInfo` object is a stub entity 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 retains all links to bulk invoices and claim associations. This permits a restored claim to remain connected to these multi-claim entities, which are outside the domain graph and therefore always in the ClaimCenter database.

Because it implements the `RootInfo` delegate, the `ClaimInfo` object 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 restored from the archive.

Claims Archiving in Guidewire ClaimCenter

Claims 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 restore 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.

Note: Guidewire provides other eligibility checks you can configure, enable or disable, and modify in the Default Group Claim Archiving Rules rule set. For example, a rule in the base configuration prevents ClaimCenter from archiving claims with open activities. However, it is possible to enable, disable, or delete this rule as your business needs require.

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 archiving plugin.*

See also

- For a discussion on how to integrate to an archive backing store, see “Archiving Integration” on page 285 in the *Integration Guide*.
- For a discussion of the `IArchiveSource` demonstration plugin implementation, see “The Archive Source Plugin” on page 286 in the *Integration Guide*.
- For more information about the criteria used by the query used for claim selection, see “Selecting Claims for Archive Eligibility” on page 668.

Archiving and Encryption

You are responsible for implementing the `IArchiveSource` plugin 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 case, this is the entire document. 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` field. Specifically, for a claim to be archivable, its `DateEligibleForArchive` field 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 not set the <code>DateEligibleForArchive</code> field as it creates a claim. Therefore, the value of the <code>DateEligibleForArchive</code> field is null.	null
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 rules sets <code>DateEligibleForArchive</code> to null.	null

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 field, 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` field on any closed claims that you load through staging tables. ClaimCenter does **not** set a `DateEligibleForArchive` date on these claims.

Restoring Claims from the Command Line

An administrator can restore one or a group of claims from the command line, by using `maintenance_tools` in `admin/bin`.

To restore a single claim, type:

```
maintenance_tools.bat -restore comment -claim claimnumber -user user -password password
```

To restore 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 171 in the *System Administration Guide*

Monitoring Claim Archiving Activity

ClaimCenter provides several different 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">• Entities archived• Entities excluded because of rules• Entities excluded because of failure The Archive Info page provides tools to reset various archive items as well. See “Info Pages” on page 159 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 **Excluded**. 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 claims that ClaimCenter successfully archived. To configure these log messages, uncomment and edit—if necessary—the `Server Archiving Success` category in file `logging.properties`. If you make changes to `logging.properties`, then (if one does not already exist), make a copy in the following directory and modify that version:

```
.../modules/cc/config/logging
```

Configuring Claims Archiving

In working with claims 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
<code>ArchiveEnabled</code>	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 also 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 Extractable entities.</p> <p>IMPORTANT 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>
<code>AssignClaimToRetriever</code>	Boolean	<p>Specifies to whom ClaimCenter assigns a restored claim:</p> <ul style="list-style-type: none">• <code>True</code> assigns the claim to the user who restored the claim.• <code>False</code> assigns a restored claim to the original group and user who owned it.
<code>DaysClosedBeforeArchive</code>	Integer	<p>Used by the <code>Claim Closed</code> rule in the base configuration to set the <code>DateEligibleForArchive</code> field 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>
<code>DaysRetrievedBeforeArchive</code>	Integer	<p>Used by the implementation of the <code>IArchiveSource</code> plugin in the base configuration to set the <code>DateEligibleForArchive</code> field on <code>Claim</code> as it retrieves a claim from the archive store.</p> <p>The default in the base configuration is 100.</p>

Parameter	Type	Description
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> <pre>relative_entity_name:foreign_key_property_name</pre>
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>
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"> One activity for the retriever of the claim One activity for the assigned user of the claim, if different from the retriever <p>The default in the base configuration is restore.</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>

See Also

- “Archive Parameters” on page 39 in the *Configuration Guide* for more details on these configuration parameters.

Archive Rules

Through the Archive rules, you can do the following:

Skip a claim 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:

```
claim.skipFromArchiving(String)
```

IMPORTANT Calling this method on a claim terminates rule set execution.

Exclude a claim 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:

```
claim.reportArchivingProblem(String)
```

Calling this method on a claim does **not** terminate rule set execution.

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"> • Approved • Check Pending Approval • Awaiting Submission <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. Note 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” on page 48 in the *Rules Guide*.

Archive Events

Every time that ClaimCenter creates a claim, it also creates a `ClaimInfo` entity. Every time a claim changes (including during claim archiving, retrieval, claim exclusion, or archive failure), ClaimCenter generates a `ClaimChanged` event. This event does not provide specific information about the archive state of the claim, however. Instead, to determine the archive state of a claim, use `ClaimInfo.ArchiveState`. This field, from the `ArchiveState` typelist, can have the following possible values in the base configuration:

Archive state	The claim is...
archived	Finished archiving
retrieving	Marked for retrieval

Thus:

- The act of archiving a claim generates a `ClaimChanged` event and a `ClaimInfo.ArchiveState` of `archived`.

- The act of retrieving a claim generates a `ClaimChanged` event with a `ClaimInfo.ArchiveState` of `retrieving`.

A null `ArchiveState` indicates that the claim is in the active ClaimCenter database. This is because it has never been archived or because ClaimCenter has successfully restored the claim from the archive.

In addition to the `ArchiveState` events, a claim `History` receives a new entry each time ClaimCenter archives or retrieves the claim. Archive rules that reference these history entries or a `ClaimChanged` event can distinguish archived and restored claims from other claims.

See also

- “Detecting Archive Events” on page 49 in the *Rules Guide*

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” on page 48 in the *Rules Guide*. See also “Archive Rules” on page 671.
- You can configure how the `Archive` writer and worker daemons behave. See “Configuring Distributed Work Queues” on page 133 in the *System Administration Guide*.
- You can view the current status of the archive process and manually control it. See “Monitoring Claim Archiving Activity” on page 669 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

- For instructions on how to gather database statistics, see “Configuring Database Statistics” on page 53 in the *System Administration Guide*.
- For more on work queues, see “Understanding Distributed Work Queues” on page 130 in the *System Administration Guide*.
- For more on batch processes and work queues, see “Batch Processes and Distributed Work Queues” on page 134 in the *System Administration Guide*

The Archive Plugin

If you implement archiving, then you must implement an archiving 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 archiving 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 archiving plugin that meets your specific business needs. Any archive plugin that you create must implement the `IArchiveSource` interface.

See also

- “Archiving Integration” on page 285 in the *Integration Guide*