

# Guidewire BillingCenter®

## BillingCenter Configuration Guide

RELEASE 8.0.4

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

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

All other trademarks are the property of their respective owners.

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

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

Product Name: Guidewire BillingCenter

Product Release: 8.0.4

Document Name: *BillingCenter Configuration Guide*

Document Revision: 02-July-2015

# Contents

<b>About BillingCenter Documentation . . . . .</b>	<b>17</b>
Conventions in This Document . . . . .	18
Support . . . . .	18

## Part I

### BillingCenter Configuration Basics

<b>1 Overview of BillingCenter Configuration . . . . .</b>	<b>21</b>
What You Can Configure . . . . .	21
How You Configure BillingCenter . . . . .	22
Types of Application Environments . . . . .	23
The Development Environment . . . . .	23
The Production Environment . . . . .	23
Deploying Configuration Files . . . . .	23
Deploying Changes in a Development Environment . . . . .	23
Deploying Changes to the Production Server . . . . .	24
Regenerating the Data Dictionary and Security Dictionary . . . . .	24
Generating the Data and Security Dictionaries in HTML Format . . . . .	25
Generating the Data and Security Dictionaries in XML Format . . . . .	25
Generating the Dictionaries as You Generate a .war or .ear File . . . . .	25
Aspects of Regenerating the Security Dictionary . . . . .	25
Managing Configuration Changes . . . . .	26
<b>2 Application Configuration Parameters . . . . .</b>	<b>27</b>
Working with Configuration Parameters . . . . .	28
Accessing Configuration Parameters in Gosu . . . . .	28
Configuration Parameter Attributes . . . . .	28
Adding Custom MIME Types . . . . .	29
Assignment Parameters . . . . .	30
AssignmentQueuesEnabled . . . . .	30
Batch Process Parameters . . . . .	30
BatchProcessHistoryPurgeDaysOld . . . . .	30
Business Calendar Parameters . . . . .	30
BusinessDayDemarcation . . . . .	30
BusinessDayEnd . . . . .	30
BusinessDayStart . . . . .	31
BusinessWeekEnd . . . . .	31
HolidayList (Obsolete) . . . . .	31
IsFridayBusinessDay . . . . .	31
IsMondayBusinessDay . . . . .	31
IsSaturdayBusinessDay . . . . .	31
IsSundayBusinessDay . . . . .	31
IsThursdayBusinessDay . . . . .	31
IsTuesdayBusinessDay . . . . .	32
IsWednesdayBusinessDay . . . . .	32
MaxAllowedDate . . . . .	32
MinAllowedDate . . . . .	32

Cache Parameters .....	32
ChargePatternCacheRefreshIntervalSecs .....	32
ExchangeRatesCacheRefreshIntervalSecs .....	32
GlobalCacheActiveTimeMinutes .....	33
GlobalCacheDetailedStats.....	33
GlobalCacheReapingTimeMinutes.....	33
GlobalCacheSizeMegabytes .....	33
GlobalCacheSizePercent .....	33
GlobalCacheStaleTimeMinutes .....	34
GlobalCacheStatsRetentionPeriodDays .....	34
GlobalCacheStatsWindowMinutes .....	34
GroupCacheRefreshIntervalSecs .....	34
PlanOrderCacheRefreshIntervalSecs .....	34
ScriptParametersRefreshIntervalSecs.....	34
TreeViewRefresh .....	35
TAccountPatternCacheRefreshInternalSecs .....	35
ZoneCacheRefreshIntervalSecs .....	35
Clustering Parameters .....	35
ClusteringEnabled.....	35
ClusterMemberPurgeDaysOld .....	36
ClusterMemberRecordUpdateIntervalSecs.....	36
ClusterMulticastAddress .....	36
ClusterMulticastPort .....	36
ClusterMulticastTTL.....	36
ClusterProtocolStack.....	37
ClusterProtocolStackOption1 .....	37
ClusterProtocolStackOption2 .....	38
ClusterStatisticsMonitorIntervalMins.....	38
ConfigVerificationEnabled .....	38
JGroupsClusterChannel .....	38
JGroupsWatchdogHeartbeatIntervalSecs .....	38
JGroupsWatchdogMissedHeartbeatsBeforeReset.....	39
PDFMergeHandlerLicenseKey .....	39
Database Parameters .....	39
DisableHashJoinForAgencyCycleException .....	39
DisableHashJoinForPolicySearch .....	39
DisableIndexFastFullScanForAgencyCycleException .....	39
DisableSortMergeJoinForAgencyCycleException .....	40
DisableSortMergeJoinForPolicySearch .....	40
DiscardQueryPlansDuringStatsUpdateBatch .....	40
IdentifyQueryBuilderViaComments .....	40
IdentifyORMLayerViaComments .....	40
MigrateToLargeIDsAndDatetime2 .....	41
Document Creation and Document Management Parameters .....	41
AllowDocumentAssistant .....	41
DisplayDocumentEditUploadButtons.....	41
DocumentAssistantJNLP .....	41
DocumentContentDispositionMode .....	41
DocumentTemplateDescriptorXSDLLocation .....	42
MaximumFileUploadSize .....	42
UseDocumentAssistantToDisplayDocuments .....	42
Domain Graph Parameters .....	42
DomainGraphKnownLinksWithIssues .....	42
DomainGraphKnownUnreachableTables .....	42

Environment Parameters.....	43
AllocateInvoiceNumberOnInit .....	43
AlwaysShowPhoneWidgetRegionCode .....	43
CollapseFutureInvoicesUponCancellation .....	43
CommissionRemainderTreatment .....	43
CreateRollupTxnsUponPolicyClosure .....	43
CurrentEncryptionPlugin .....	44
DaysBeforeAccountIsConsideredInactive .....	44
EnableChargeProRataTxCreation .....	44
EnableInternalDebugTools .....	45
GosuSampleDataMethod .....	45
IBillingCenterAPICDCENumRetries.....	45
IncludeClosedPoliciesInAccountAggregateBalances.....	45
InvoicePeriodCutoff .....	45
KeyGeneratorRangeSize .....	45
MemoryUsageMonitorIntervalMins .....	45
PASEffectiveTime .....	45
PublicIDPrefix .....	46
ResourcesMutable.....	46
StrictDataTypes.....	46
UnrestrictedUserName .....	47
Financial Parameters.....	47
ProRataCalculationRemainderTreatment .....	47
Globalization Parameters .....	48
DefaultApplicationLanguage .....	48
DefaultApplicationLocale .....	48
DefaultApplicationCurrency .....	49
DefaultRoundingMode .....	49
MulticurrencyDisplayMode .....	50
DefaultCountryCode .....	50
DefaultPhoneCountryCode .....	50
DefaultNANPACountryCode .....	50
AlwaysShowPhoneWidgetRegionCode .....	50
Integration Parameters .....	50
DefaultXmlExportIEncryptionId .....	51
LockPrimaryEntityDuringMessageHandling .....	51
PluginStartupTimeout .....	51
Logging Parameters .....	51
LoggerCategorySource .....	52
LoggersShowLog4j .....	52
LoggersShowPredefined .....	52
Miscellaneous Parameters.....	52
ConsistencyCheckerThreads .....	52
EquityDatingIncludeWriteoffs .....	53
ListViewPageSizeDefault .....	53
MaxTAccountsInConsistencyCheck .....	53
MaxTransactionsInConsistencyCheck .....	53
ProfilerDataPurgeDaysOld .....	53
TransactionIdPurgeDaysOld .....	53

PDF Print Settings Parameters .....	53
DefaultContentDispositionMode .....	53
PrintFontFamilyName .....	54
PrintFontSize .....	54
PrintFOPUserConfigFile .....	54
PrintHeaderFontSize .....	54
PrintLineHeight .....	54
PrintListViewBlockSize .....	54
PrintListViewFontSize .....	54
PrintMarginBottom .....	55
PrintMarginLeft .....	55
PrintMarginRight .....	55
PrintMarginTop .....	55
PrintMaxPDFInputFileSize .....	55
PrintPageHeight .....	55
PrintPageWidth .....	55
Scheduler and Workflow Parameters .....	56
SchedulerEnabled .....	56
WorkflowLogDebug .....	56
WorkflowLogPurgeDaysOld .....	56
WorkflowPurgeDaysOld .....	56
WorkflowStatsIntervalMins .....	56
Search Parameters .....	56
MaxContactSearchResults .....	57
MaxSearchResults .....	57
Security Parameters .....	57
EnableDownlinePermissions .....	57
FailedAttemptsBeforeLockout .....	57
LockoutPeriod .....	58
LoginRetryDelay .....	58
MaxPasswordLength .....	58
MinPasswordLength .....	58
RestrictContactPotentialMatchToPermittedItems .....	58
RestrictSearchesToPermittedItems .....	58
SessionTimeoutSecs .....	58
User Interface Parameters .....	59
ActionsShortcut .....	59
AutoCompleteLimit .....	59
InputMaskPlaceholderCharacter .....	59
ListViewPageSizeDefault .....	59
MaxBrowserHistoryItems (Obsolete) .....	59
QuickJumpShortcut .....	59
UISkin .....	59
WizardNextShortcut .....	59
WizardPrevShortcut .....	60
WizardPrevNextButtonsVisible .....	60

Work Queue Parameters .....	60
EnableWorkQueueValidation .....	60
InstrumentedWorkerInfoPurgeDaysOld .....	60
WorkItemCreationBatchSize .....	60
WorkItemPriorityMultiplierSecs .....	60
WorkItemRetryLimit .....	61
WorkQueueHistoryMaxDownload .....	61
WorkQueueThreadPoolMaxSize .....	61
WorkQueueThreadPoolMinSize .....	61
WorkQueueThreadsKeepAliveTime .....	62

## Part II

### The Guidewire Development Environment

<b>3 Getting Started .....</b>	<b>65</b>
What Is Guidewire Studio? .....	65
The Studio Development Environment .....	66
Working with the QuickStart Development Server .....	67
Connecting the Development Server to a Database .....	68
Deploying Your Configuration Changes .....	69
BillingCenter Configuration Files .....	69
Key Directories .....	70
Using Studio with IntelliJ IDEA Ultimate Edition .....	70
Studio and the DCE VM .....	70
Starting Guidewire Studio .....	71
Restarting Studio .....	72
Using the Studio Interface .....	72
<b>4 Working in Guidewire Studio .....</b>	<b>75</b>
Entering Valid Code .....	75
The Code Menu .....	76
Using Dot Completion .....	77
Using SmartHelp .....	78
Accessing Reference Information .....	79
Accessing the Gosu API Reference .....	79
Accessing the PCF Reference Guide .....	79
Accessing the Gosu Reference Guide .....	80
Using Studio Keyboard Shortcuts .....	80
Gosu Editor .....	80
Gosu Tester .....	85
PCF Editor .....	85
Viewing Keyboard Shortcuts in BillingCenter .....	86
Using Text Editing Commands .....	87
Navigating Tables .....	87
Refactoring Gosu Code .....	88
Renaming a Gosu Resource .....	88
Moving a Gosu Resource .....	88
Saving Your Work .....	89
Validating Studio Resources .....	89
<b>5 Working with Guidewire Studio .....</b>	<b>91</b>
Improving Studio Performance .....	91
Improving Performance of Code Compilation .....	91
Increasing Memory Available to Studio .....	92
Setting Font Display Options .....	93

<b>6 BillingCenter Studio and Gosu .....</b>	<b>95</b>
Gosu Building Blocks.....	95
Gosu Case Sensitivity .....	96
Working with Gosu in BillingCenter Studio .....	96
Gosu Packages .....	97
Gosu Classes.....	97
BillingCenter Base Configuration Classes .....	98
Class Visibility in Studio .....	99
Preloading Gosu Classes.....	100
Gosu Enhancements .....	100
The Guidewire XML Model.....	101
Script Parameters .....	101
Script Parameters Overview .....	102
Working with Script Parameters.....	102
Referencing a Script Parameter in Gosu.....	103

## Part III

### Guidewire Studio Editors

<b>7 Using the Studio Editors .....</b>	<b>107</b>
Editing in Guidewire Studio .....	107
Working in the Gosu Editor .....	108
<b>8 Using the Plugins Registry Editor.....</b>	<b>109</b>
What Are Plugins? .....	109
Plugin Implementation Classes.....	109
What is the Plugins Registry?.....	109
Startable Plugins .....	110
Working with Plugins.....	110
Creating a Plugins Registry Item .....	110
Adding an Implementation to a Plugins Registry Item.....	110
Setting Environment and Server Context for Plugin Implementations.....	112
Customizing Plugin Functionality .....	112
Working with Plugin Versions .....	112
<b>9 Working with Web Services.....</b>	<b>115</b>
Web Services and Guidewire Studio .....	115
Using the Web Service Editor .....	116
Defining a Web Service Collection.....	116
<b>10 Implementing QuickJump Commands .....</b>	<b>119</b>
What Is QuickJump?.....	119
Adding a QuickJump Navigation Command .....	120
Implementing QuickJumpCommandRef Commands .....	120
Implementing StaticNavigationCommandRef Commands.....	122
Implementing ContextualNavigationCommandRef Commands .....	122
Checking Permissions on QuickJump Navigation Commands .....	122
<b>11 Using the Entity Names Editor.....</b>	<b>125</b>
Entity Names Editor .....	125
Variable Table.....	126
The Entity Path Column .....	126
The Use Entity Names? Column.....	127
The Sort Columns .....	127
Gosu Text Editor.....	128
Including Data from Subentities.....	128

Entity Name Types .....	129
<b>12 Using the Messaging Editor.....</b>	<b>131</b>
Messaging Editor .....	131
Adding a Messaging Environment .....	131
Adding a Message Destination .....	132
Associating Event Names with a Message Destination .....	134
<b>13 Using the Display Keys Editor.....</b>	<b>135</b>
Display Keys Editor .....	135
Creating Display Keys in a Gosu Editor.....	136
Retrieving the Value of a Display Key.....	136

## Part IV Data Model Configuration

<b>14 Working with the Data Dictionary .....</b>	<b>141</b>
What is the Data Dictionary? .....	141
What Can You View in the Data Dictionary? .....	142
Using the Data Dictionary .....	142
Field Colors.....	143
Object Attributes.....	143
Entity Subtypes.....	144
Data Column and Field Types .....	144
Virtual Properties on Data Entities .....	145
<b>15 The BillingCenter Data Model .....</b>	<b>147</b>
What is the Data Model? .....	147
The Data Model in Guidewire Application Architecture .....	148
The Base Data Model .....	148
Working with Dot Notation .....	148
Overview of Data Entities.....	149
Data Entity Metadata Files .....	149
Working with Entity Definitions.....	151
Search for an Existing Entity Definition.....	152
Create a New Entity Definition.....	152
Extend an Existing Entity Definition .....	152
BillingCenter Data Entities .....	153
Data Entities and the Application Database .....	153
BillingCenter Database Tables .....	155
Data Objects and Scriptability .....	156
Base BillingCenter Data Objects .....	158
Delegate Data Objects.....	158
Entity Data Objects .....	161
Extension Data Objects .....	166
Non-persistent Entity Data Objects.....	167
Subtype Data Objects .....	169
View Entity Data Objects .....	171
View Entity Extension Data Objects.....	173

Data Object Subelements .....	174
<array> .....	176
<column> .....	178
<edgeForeignKey> .....	182
<events> .....	185
<foreignkey> .....	186
<fulldescription> .....	188
<implementsEntity> .....	188
<implementsInterface> .....	189
<index> .....	190
<onetoone> .....	191
<remove-index> .....	193
<tag> .....	194
<typekey> .....	194
<b>16 Working with Associative Arrays .....</b>	<b>197</b>
Overview of Associative Arrays .....	197
Associative Array Mapping Types .....	198
Scriptability and Associative Arrays .....	198
Issues with Setting Array Member Values .....	199
Subtype Mapping Associative Arrays .....	199
Working with Array Values Using Subtype Mapping .....	200
Typelist Mapping Associative Arrays .....	201
Working with Array Values Using Typelist Mapping .....	202
<b>17 Modifying the Base Data Model .....</b>	<b>205</b>
Planning Changes to the Base Data Model .....	205
Overview of Data Model Extension .....	205
Strategies for Extending the Base Data Model .....	206
What Happens If You Change the Data Model? .....	207
Naming Restrictions for Extensions .....	208
Defining a New Data Entity .....	208
Extending a Base Configuration Entity .....	209
Working with Attribute Overrides .....	210
Extending the Base Data Model: Examples .....	212
Creating a New Delegate Object .....	212
Extending a Delegate Object .....	214
Defining a Subtype .....	215
Defining a Reference Entity .....	216
Defining an Entity Array .....	216
Implementing a Many-to-Many Relationship Between Entity Types .....	217
Extending an Existing View Entity .....	218
Removing Objects from the Base Configuration Data Model .....	219
Removing a Base Extension Entity .....	220
Removing an Extension to a Base Object .....	221
Implications of Modifying the Data Model .....	221
Deploying Data Model Changes to the Application Server .....	223
<b>18 Data Types .....</b>	<b>225</b>
Overview of Data Types .....	225
Working with Data Types .....	226
Using Data Types .....	226
Defining a Data Type for a Property .....	227

The Data Types Configuration File .....	228
<...DataType> .....	228
Deploying Modifications to Data Types Configuration File .....	228
Customizing Base Configuration Data Types .....	229
List of Customizable Data Types .....	230
Working with the Medium Text Data Type (Oracle) .....	231
The Data Type API .....	231
Retrieving the Data Type for a Property .....	232
Retrieving a Particular Data Type in Gosu .....	232
Retrieving a Data Type Reflectively .....	232
Using the IDataType Methods .....	232
Defining a New Data Type: Required Steps .....	233
Defining a New Tax Identification Number Data Type .....	233
Step 1: Register the Data Type .....	234
Step 2: Implement the IDataTypeDef Interface .....	234
Step 3: Implement the Data Type Aspect Handlers .....	235
<b>19 Field Validation .....</b>	<b>239</b>
Field Validators .....	239
Field Validator Definitions .....	240
<FieldValidators> .....	241
<ValidatorDef> .....	241
Modifying Field Validators .....	243
Using <columnOverride> to Modify Field Validation .....	243
<b>20 Working with Typelists .....</b>	<b>245</b>
What is a Typelist? .....	246
Terms Related to Typelists .....	246
Typelists and Typecodes .....	246
Typelist Definition Files .....	247
Different Kinds of Typelists .....	248
Internal Typelists .....	248
Extendable Typelists .....	249
Custom Typelists .....	249
Working with Typelists in Studio .....	249
The Typelists Editor .....	249
Entering Typecodes .....	251
Typekey Fields .....	252
Removing or Retiring a Typekey .....	254
Removing a Typekey .....	255
Typelist Filters .....	255
Static Filters .....	256
Creating a Static Filter Using Categories .....	257
Creating a Static Filter Using Includes .....	258
Creating a Static Filter Using Excludes .....	259
Dynamic Filters .....	260
Creating a Dynamic Filter .....	261
Typecode References in Gosu .....	263
Mapping Typecodes to External System Codes .....	264
<b>Part V</b>	
<b>User Interface Configuration</b>	
<b>21 Using the PCF Editor .....</b>	<b>269</b>
Page Configuration (PCF) Editor .....	269

Page Canvas Overview .....	270
Creating a New PCF File .....	270
Working with Shared or Included Files .....	271
Understanding PCF Modes .....	272
Setting a PCF Mode .....	272
Creating New Modal PCF files .....	273
Page Config Menu .....	273
Toolbox Tab .....	274
Structure Tab .....	274
Properties Tab .....	275
Child Lists .....	276
PCF Elements .....	277
PCF Elements and the Properties Tab .....	277
Working with Elements .....	277
Adding an Element to the Canvas .....	278
Moving an Element on the Canvas .....	278
Changing the Type of an Element .....	279
Adding a Comment to an Element .....	279
Finding an Element on the Canvas .....	280
Viewing the Source of an Element .....	280
Duplicating an Element .....	280
Deleting an Element .....	280
Copying an Element .....	281
Cutting an Element .....	281
Pasting an Element .....	281
Linking Widgets .....	281
<b>22 Introduction to Page Configuration.....</b>	<b>283</b>
Page Configuration Files .....	283
Page Configuration Elements .....	283
What is a PCF Element? .....	284
Types of PCF Elements .....	284
Identifying PCF Elements in the User Interface .....	286
Getting Started Configuring Pages .....	288
Finding an Existing Element To Edit .....	288
Creating a New Standalone PCF Element .....	289
Modifying Style and Theme Elements .....	290
Changing or Adding Images .....	290
Overriding CSS .....	290
Changing Theme Colors .....	290
Advanced Re-Theming .....	291
<b>23 Data Panels .....</b>	<b>293</b>
Panel Overview .....	293
Detail View Panel .....	293
Define a Detail View .....	294
Add Columns to a Detail View .....	295
Format a Detail View .....	296
List View Panel .....	298
Define a List View .....	299
Iterate a List View Over a Data Set .....	301
Choose the Data Source for a List View .....	302
<b>24 Location Groups .....</b>	<b>305</b>
Location Group Overview .....	305

Define a Location Group .....	306
Location Groups as Navigation .....	307
Location Groups as Tab Menus .....	307
Location Groups as Menu Links .....	308
Location Groups as Screen Tabs .....	308
<b>25 Navigation .....</b>	<b>311</b>
Navigation Overview .....	311
Tab Bars .....	312
Configure the Default Tab Bar .....	312
Specify Which Tab Bar to Display .....	312
Define a Tab Bar .....	313
Tabs .....	313
Define a Tab .....	313
Define a Drop-down Menu on a Tab .....	313
<b>26 Configuring Search Functionality .....</b>	<b>315</b>
BillingCenter Search Functionality .....	315
Configuring BillingCenter Search .....	316
Limiting Search Results .....	317
Working with Search Criteria in XML .....	317
The <CriteriaDef> Element .....	318
The <Criterion> Subelement .....	319
Performance Tuning for Specific Search Criteria .....	320
Do Not Attempt to Modify the Required Search Properties .....	320
Working with Search Criteria in Gosu .....	320
Example: Modifying the PolicySearchCriteria Class .....	321
Enabling Searches on the Address Field .....	322
The SearchMethods Class .....	323
Search Criteria Validation Upon Server Start-up .....	323
<b>27 Configuring Special Page Functions .....</b>	<b>325</b>
Adding Print Capabilities .....	325
Overview of the Print Functionality .....	325
List View Printing .....	327
Linking to a Specific Page: Using an EntryPoint PCF .....	327
Entry Points .....	327
Creating a Forwarding EntryPoint PCF .....	329
Linking to a Specific Page: Using an ExitPoint PCF .....	329
Creating an ExitPoint PCF .....	329
<b>Part VI</b>	
<b>Workflow and Activity Configuration</b>	
<b>28 Using the Workflow Editor .....</b>	<b>335</b>
Workflow in Guidewire BillingCenter .....	335
Workflow in Guidewire Studio .....	336
Understanding Workflow Steps .....	337
Using the Workflow Right-Click Menu .....	338
Using Search with Workflow .....	338

<b>29 Guidewire Workflow .....</b>	<b>341</b>
Understanding Workflow .....	341
Workflow Instances .....	342
Work Items .....	343
Workflow Process Format.....	343
Workflow Step Summary .....	343
Workflow Gosu.....	344
Workflow Versioning .....	344
Workflow Localization .....	346
Workflow Structural Elements .....	346
<Context> .....	346
<Start>.....	347
<Finish> .....	347
Common Step Elements .....	347
Enter and Exit Scripts .....	347
Asserts.....	348
Events .....	348
Notifications .....	348
Branch IDs .....	349
Basic Workflow Steps .....	349
AutoStep .....	349
MessageStep .....	350
ActivityStep .....	351
ManualStep.....	352
Outcome .....	353
Step Branches .....	354
Working with Branch IDs.....	355
GO.....	355
TRIGGER .....	356
TIMEOUT.....	358
Creating New Workflows .....	359
Cloning an Existing Workflow .....	359
Extending an Existing Workflow .....	359
Extending a Workflow: A Simple Example .....	360
Instantiating a Workflow .....	363
A Simple Example of Instantiation.....	363
The Workflow Engine .....	365
Distributed Execution .....	365
Synchronicity, Transactions, and Errors.....	366
Workflow Subflows .....	368
Workflow Administration.....	369
Workflow Debugging, Logging, and Testing.....	370
<b>30 Defining Activity Patterns .....</b>	<b>373</b>
What is an Activity Pattern?.....	373
Pattern Types and Categories .....	374
Activity Pattern Types .....	374
Categorizing Activity Patterns .....	374
Using Activity Patterns in Gosu .....	375
Calculating Activity Due Dates .....	376
Target Due Dates (Deadlines).....	376
Escalation Dates .....	376
Configuring Activity Patterns.....	376
Using Activity Patterns with Documents and Emails.....	378

Localizing Activity Patterns .....	378
------------------------------------	-----

## Part VII Testing Gosu Code

<b>31 Testing and Debugging Your Configuration .....</b>	<b>383</b>
Testing BillingCenter With Guidewire Studio .....	383
Running BillingCenter Without Debugging .....	384
Debugging BillingCenter Within Studio .....	384
Debugging a BillingCenter Server That Is Running Outside of Studio .....	384
The Studio Debugger .....	386
Setting Breakpoints .....	386
Stepping Through Code .....	387
Viewing Current Values .....	388
Enabling the Viewing of Entities While Debugging .....	388
Viewing Variables .....	388
Defining a Watch List .....	388
Resuming Execution .....	389
Using the Gosu Scratchpad .....	389
Executing Code in the Gosu Scratchpad .....	389
Accessing Application Data in the Gosu Scratchpad .....	390
Suggestions for Testing Rules .....	390
<b>32 Using GUnit .....</b>	<b>391</b>
The TestBase Class .....	391
Overriding TestBase Methods .....	392
Configuring the Server Environment .....	393
Configuring the Test Environment .....	394
Configuration Parameters .....	395
Creating a GUnit Test Class .....	396
Using Entity Builders to Create Test Data .....	398
Creating an Entity Builder .....	399
Entity Builder Examples .....	401
Creating New Builders .....	404

## Part VIII Guidewire BillingCenter Configuration

<b>33 BillingCenter Workflows and Delinquency Plans .....</b>	<b>413</b>
Workflows in BillingCenter .....	413
BillingCenter Workflows and Delinquency Events .....	414
Delinquency Events in Studio .....	415
Delinquency Events in BillingCenter .....	415
BillingCenter Delinquency Event Fields .....	416
Adding Delinquency Events to a BillingCenter Delinquency Plan .....	417
Creating a New Delinquency Plan: An Example .....	418
<b>34 Configuring the Charge Invoicing Process .....</b>	<b>423</b>
The Charge Invoicing Process for New Charges .....	423
Expected Results and Plugin Customizations .....	427
Modifying an Existing Charge .....	427
The ChargeInstallmentChanger Class .....	427
Removing an Entry Object .....	428

Extension Properties for Charge and InvoiceItem Objects . . . . .	429
Non-Array Type Extension Properties . . . . .	430
Array Type Extension Properties . . . . .	430
Charge Invoicing and Payment Plans . . . . .	432
Payment Plan Modifiers . . . . .	433
Processing Payment Plan Modifiers in BillingCenter . . . . .	433
Accessing Payment Plan Modifiers . . . . .	433
InvoiceStream and DateSequence Plugins . . . . .	434
<b>35 Configuring Payment Allocation Plans . . . . .</b>	<b>437</b>
Payment Allocation Plan Configuration Overview . . . . .	437
Writing an Invoice Item Filter . . . . .	437
Programming Tasks . . . . .	437
Sample Invoice Item Filter in Gosu . . . . .	438
Ordering of Invoice Items . . . . .	439
Programming Tasks . . . . .	440
Sample Ordering of Invoice Items in Gosu . . . . .	440
Modifying the DirectBillPayment Plugin . . . . .	441
The allocatePayment Method . . . . .	442
<b>36 Configuring Distributions . . . . .</b>	<b>443</b>
Modifying Direct Bill Payment and Credit Distributions . . . . .	444
Direct Bill Payment Object Model . . . . .	444
Example Modification of a Direct Bill Payment and Credit Distribution . . . . .	444
Modifying Agency Bill Payment and Credit Distributions . . . . .	445
Modifying Agency Bill Promise Distributions . . . . .	446
<b>37 Configuring Write-Offs . . . . .</b>	<b>449</b>
Commission Write-Offs . . . . .	449
Producer Write-Offs . . . . .	450
Defining a New Creation Date Filter . . . . .	450
<b>38 Configuring Credit Handling . . . . .</b>	<b>451</b>
Defining New Return Premium Plan Property Settings . . . . .	451
Defining New Excess Treatment Settings . . . . .	452
Defining a New Excess Treatment Setting: Disbursement . . . . .	452
Defining a New Excess Treatment Setting: Suspense Item . . . . .	453
Identifying Eligible Invoice Items . . . . .	454
Configuring Credit Allocation . . . . .	454
Defining a New Credit Allocation Method . . . . .	455
<b>39 Configuring Multicurrency . . . . .</b>	<b>459</b>
Configuring BillingCenter with a Single Currency . . . . .	459
Configuring BillingCenter with Multiple Currencies . . . . .	459
Currency Silos . . . . .	460
Multicurrency Delegates . . . . .	460
Monetary Data Types . . . . .	460
Exceptions to Currency Silo Rules . . . . .	461
Enabling Multicurrency Integration . . . . .	461
Set up Currency-Specific Plans and Authority Limits . . . . .	461

# About BillingCenter Documentation

The following table lists the documents in BillingCenter documentation.

Document	Purpose
<i>InsuranceSuite Guide</i>	If you are new to Guidewire InsuranceSuite applications, read the <i>InsuranceSuite Guide</i> for information on the architecture of Guidewire InsuranceSuite and application integrations. The intended readers are everyone who works with Guidewire applications.
<i>Application Guide</i>	If you are new to BillingCenter or want to understand a feature, read the <i>Application Guide</i> . This guide describes features from a business perspective and provides links to other books as needed. The intended readers are everyone who works with BillingCenter.
<i>Upgrade Guide</i>	Describes how to upgrade BillingCenter from a previous major version. The intended readers are system administrators and implementation engineers who must merge base application changes into existing BillingCenter application extensions and integrations.
<i>New and Changed Guide</i>	Describes new features and changes from prior BillingCenter versions. Intended readers are business users and system administrators who want an overview of new features and changes to features. Consult the "Release Notes Archive" part of this document for changes in prior maintenance releases.
<i>Installation Guide</i>	Describes how to install BillingCenter. The intended readers are everyone who installs the application for development or for production.
<i>System Administration Guide</i>	Describes how to manage a BillingCenter system. The intended readers are system administrators responsible for managing security, backups, logging, importing user data, or application monitoring.
<i>Configuration Guide</i>	The primary reference for configuring initial implementation, data model extensions, and user interface (PCF) files. The intended readers are all IT staff and configuration engineers.
<i>Globalization Guide</i>	Describes how to configure BillingCenter for a global environment. Covers globalization topics such as global regions, languages, date and number formats, names, currencies, addresses, and phone numbers. The intended readers are configuration engineers who localize BillingCenter.
<i>Rules Guide</i>	Describes business rule methodology and the rule sets in BillingCenter Studio. The intended readers are business analysts who define business processes, as well as programmers who write business rules in Gosu.
<i>Contact Management Guide</i>	Describes how to configure Guidewire InsuranceSuite applications to integrate with ContactManager and how to manage client and vendor contacts in a single system of record. The intended readers are BillingCenter implementation engineers and ContactManager administrators.
<i>Best Practices Guide</i>	A reference of recommended design patterns for data model extensions, user interface, business rules, and Gosu programming. The intended readers are configuration engineers.
<i>Integration Guide</i>	Describes the integration architecture, concepts, and procedures for integrating BillingCenter with external systems and extending application behavior with custom programming code. The intended readers are system architects and the integration programmers who write web services code or plugin code in Gosu or Java.
<i>Gosu Reference Guide</i>	Describes the Gosu programming language. The intended readers are anyone who uses the Gosu language, including for rules and PCF configuration.
<i>Glossary</i>	Defines industry terminology and technical terms in Guidewire documentation. The intended readers are everyone who works with Guidewire applications.

## Conventions in This Document

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

## Support

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

---

part I

# BillingCenter Configuration Basics



# Overview of BillingCenter Configuration

This topic provides some basic information, particularly about the Guidewire BillingCenter 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 21
- “How You Configure BillingCenter” on page 22
- “Types of Application Environments” on page 23
- “Deploying Configuration Files” on page 23
- “Regenerating the Data Dictionary and Security Dictionary” on page 24
- “Managing Configuration Changes” on page 26

## What You Can Configure

Application configuration files determine virtually every aspect of the BillingCenter 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 BillingCenter Application**

You can extend the functionality of the application:

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

### **Modify the BillingCenter Interface**

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

### **Implement Security Policies**

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

### **Create Business Rules and Processes**

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

### **Designate Activity Patterns**

You can group work activities and assign to producers.

### **Integrate with External Systems**

You can integrate BillingCenter data with external applications.

### **Define Application Parameters**

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

### **Define Workflows**

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

## **How You Configure BillingCenter**

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

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

### **See also**

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

## Types of Application Environments

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

- The Development Environment
- The Production Environment

### The Development Environment

The development environment for BillingCenter has the following characteristics:

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

Guidewire provides a development environment (Guidewire Studio) that is separate from the production environment. Guidewire Studio is a stand-alone development application that runs independently of Guidewire BillingCenter. 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 BillingCenter has the following characteristics:

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

## Deploying Configuration Files

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

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

### Deploying Changes in a Development Environment

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

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

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

It is possible to use a different development environment and database other than that provided by Guidewire in the base configuration. If you do so, then deployment of modified configuration files can require additional work. For details on implementing a different development environment, see “Selecting an Installation Scenario” on page 8 in the *Installation Guide*.

## Deploying Changes to the Production Server

To deploy configuration changes to the BillingCenter 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 BillingCenter instance on the production application server.
- Deploy the .war (or .ear) file to the production application server.
- Restart the production application server.

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

### To deploy a .war (.ear) file

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

## Regenerating the Data Dictionary and Security Dictionary

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

**See also**

- To understand the *Data Dictionary* and the information it includes, see “Working with the Data Dictionary” on page 141.
- To understand the *Security Dictionary* and the information it includes, see “Security Dictionary” on page 389 in the *Application Guide*.

## Generating the Data and Security Dictionaries in HTML Format

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

```
gwbc regen-dictionary
```

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

```
BillingCenter/build/dictionary/data  
BillingCenter/build/dictionary/security
```

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

## Generating the Data and Security Dictionaries in XML Format

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

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

```
gwbc regen-dictionary -DoutputFormat=xml
```

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

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

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

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

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

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

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

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

**See also**

- For information on the Guidewire command line tools for use in a development environment, see “Commands Reference” on page 97 in the *Installation Guide*.

## Aspects of Regenerating the Security Dictionary

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

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

BillingCenter does not write this file information to the database.

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

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

---

## Managing Configuration Changes

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

# Application Configuration Parameters

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

This topic includes:

- “Working with Configuration Parameters” on page 28
- “Assignment Parameters” on page 30
- “Batch Process Parameters” on page 30
- “Business Calendar Parameters” on page 30
- “Cache Parameters” on page 32
- “Clustering Parameters” on page 35
- “Database Parameters” on page 39
- “Document Creation and Document Management Parameters” on page 41
- “Domain Graph Parameters” on page 42
- “Environment Parameters” on page 43
- “Financial Parameters” on page 47
- “Globalization Parameters” on page 48
- “Integration Parameters” on page 50
- “Logging Parameters” on page 51
- “Miscellaneous Parameters” on page 52
- “PDF Print Settings Parameters” on page 53
- “Scheduler and Workflow Parameters” on page 56
- “Search Parameters” on page 56
- “Security Parameters” on page 57
- “User Interface Parameters” on page 59
- “Work Queue Parameters” on page 60

## Working with Configuration Parameters

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

`BillingCenter/modules/configuration/config/`

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

---

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

---

This file generally contains entries in the following format:

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

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

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

### Adding Custom Parameters to BillingCenter

You cannot add new or additional configuration parameters to `config.xml`. Guidewire does not support any attempt to do so. If you want to add custom parameters to your configuration of BillingCenter, consider defining script parameters. You can access the values of script parameters in Gosu code at runtime. For more information, see “Script Parameters” on page 101.

## Accessing Configuration Parameters in Gosu

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

```
gw.api.system.PLConfigParameters  
gw.api.system.BCConfigParameters
```

For example:

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

## Configuration Parameter Attributes

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

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

### Required

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

### Set for Environment

Guidewire designates certain configuration parameters as `localok`. This indicates that it is possible for this value to vary on different servers in the same environment. For example, you can set `ClusterProtocolStackOption2` 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. BillingCenter 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 BillingCenter, see “Defining the Application Server Environment” on page 14 in the *System Administration Guide*.

### Development Environment Only

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

### Permanent

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

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

## Adding Custom MIME Types

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

## Assignment Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

### AssignmentQueuesEnabled

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

**Default:** `false`

## Batch Process Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

### BatchProcessHistoryPurgeDaysOld

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

**Default:** `45`

## Business Calendar Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

### BusinessDayDemarcation

The point in time at which a business day begins.

**Default:** `12:00 AM`

**Set For Environment:** Yes

### BusinessDayEnd

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

**Default:** `5:00 PM`

**Set For Environment:** Yes

## BusinessDayStart

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

**Default:** 8:00 AM

**Set For Environment:** Yes

## BusinessWeekEnd

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

**Default:** Friday

**Set For Environment:** Yes

## HolidayList (Obsolete)

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

## IsFridayBusinessDay

Indicates whether Friday is a business day.

**Default:** True

**Set for Environment:** Yes

## IsMondayBusinessDay

Indicates whether Monday is a business day.

**Default:** True

**Set for Environment:** Yes

## IsSaturdayBusinessDay

Indicates whether Saturday is a business day.

**Default:** False

**Set for Environment:** Yes

## IsSundayBusinessDay

Indicates whether Sunday is a business day.

**Default:** False

**Set for Environment:** Yes

## IsThursdayBusinessDay

Indicates whether Thursday is a business day.

**Default:** True

**Set for Environment:** Yes

## IsTuesdayBusinessDay

Indicates whether Tuesday is a business day.

**Default:** True

**Set for Environment:** Yes

## IsWednesdayBusinessDay

Indicates whether Wednesday is a business day.

**Default:** True

**Set for Environment:** Yes

## MaxAllowedDate

The latest date allowed to be used.

**Default:** 2200-12-31

**Set For Environment:** Yes

## MinAllowedDate

The earliest date allowed to be used.

**Default:** 1800-01-01

**Set For Environment:** Yes

# Cache Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

### See also

- “Application Server Caching” on page 63 in the *System Administration Guide*

## ChargePatternCacheRefreshIntervalSecs

The time between refreshes of the charge pattern cache, in seconds. This integer value must be 0 or greater.

**Default:** 3600

## ExchangeRatesCacheRefreshIntervalSecs

The time between refreshes of the ExchangeRateSet cache, in seconds. This integer value must be 0 or greater. See “Application Server Caching” on page 63 in the *System Administration Guide* for more information.

**Default:** 600

## GlobalCacheActiveTimeMinutes

Time period (in minutes) in which BillingCenter 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 BillingCenter is using one or more items very heavily. For example, this can be the length of time that a user stays on a page. The maximum value for this parameter is the smaller of 15 and the value of `GlobalCacheStaleTimeMinutes`.

See “Application Server Caching” on page 63 in the *System Administration Guide* for more information.

**Default:** 15

**Minimum:** 1

**Maximum:** 15

**Set for Environment:** Yes

## GlobalCacheDetailedStats

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

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

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

**Default:** False

## GlobalCacheReapingTimeMinutes

Time (in minutes) since the last use before BillingCenter considers cached items to be eligible for reaping. You can think of this as the period during which BillingCenter is most likely to reuse an entity. See “Application Server Caching” on page 63 in the *System Administration Guide* for more information.

**Default:** 15

**Minimum:** 1

**Maximum:** 15

**Set for Environment:** Yes

## GlobalCacheSizeMegabytes

The amount of space to allocate to the global cache. If you specify this value, it takes precedence over `GlobalCacheSizePercent`. See “Application Server Caching” on page 63 in the *System Administration Guide* for more information.

**Null:** Yes

**Set for Environment:** Yes

## GlobalCacheSizePercent

The percentage of the heap to allocate to the global cache. See “Application Server Caching” on page 63 in the *System Administration Guide* for more information.

**Default:** 15

**Set for Environment:** Yes

## GlobalCacheStaleTimeMinutes

Time (in minutes) since the last write before BillingCenter considers cached items to be stale and thus eligible for reaping. See “Application Server Caching” on page 63 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 BillingCenter preserves statistics for historical comparison. See “Application Server Caching” on page 63 in the *System Administration Guide* for more information.

**Default:** 7

**Minimum:** 2

**Maximum:** 365

**Set for Environment:** Yes

## GlobalCacheStatsWindowMinutes

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

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

See “Application Server Caching” on page 63 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 0 or greater. See “Application Server Caching” on page 63 in the *System Administration Guide* for more information.

**Default:** 600

## PlanOrderCacheRefreshIntervalSecs

The time between refreshes of the plan order cache, in seconds. This integer value must be 0 or greater.

**Default:** 600

## ScriptParametersRefreshIntervalSecs

The time between refreshes of the script parameter cache, in seconds. This integer value must be 0 or greater. See “Application Server Caching” on page 63 in the *System Administration Guide* for more information.

**Default:** 600

## TreeViewRefresh

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

**Default:** 120

## TAccountPatternCacheRefreshInternalSecs

The time between refreshes of the charge pattern cache, in seconds. This integer value must be 0 or greater.

**Default:** 3600

## ZoneCacheRefreshIntervalSecs

The time between refreshes of the zone cache, in seconds. See “Application Server Caching” on page 63 in the *System Administration Guide* for more information.

**Default:** 86400

# Clustering Parameters

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

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

### See also

- “Clustering Application Servers” on page 73 in the *System Administration Guide*

## ClusteringEnabled

Whether to enable clustering on this application server.

### Studio Read-only Mode

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

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

To indicate the read-only status:

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

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

**Default:** `False`

**Set for Environment:** Yes

## ClusterMemberPurgeDaysOld

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

**Default:** 30

## ClusterMemberRecordUpdateIntervalSecs

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

**Default:** 60

## ClusterMulticastAddress

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

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

224.0.0.0 – 239.255.255.255

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

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

**Default:** 228.9.9.9

**Set for Environment:** Yes

## ClusterMulticastPort

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

**Default:** 45566

**Set for Environment:** Yes

## ClusterMulticastTTL

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

**Default:** 1

**Set for Environment:** Yes

## ClusterProtocolStack

JGroups configuration template to use with multicast UDP communications. Variables in the template refer to other cluster configuration parameters:

```
UDP(mcast_addr=${multicastAddress};ip_ttl=${timeToLive};mcast_port=${port}${option1}):
```

BillingCenter replaces the variables in ClusterProtocolStack with the values of the following configuration parameters:

ClusterProtocolStack variable	Replaced with the value of...
\${multicastAddress}	ClusterMulticastAddress
\${timeToLive}	ClusterMulticastTTL
\${port}	ClusterMulticastPort
\${option1}	ClusterProtocolStackOption1

Guidewire recommends that you do not change the default value of ClusterProtocolStack. Modify the values of the listed cluster configuration parameters, instead.

If you use TCP communications, rather than UDP, you can override the value of this configuration parameter completely. See “Considerations for a Clustered Application Server Environment” on page 16 in the *Installation Guide* for a discussion on the use of TCP as the cluster network protocol.

See the following:

- ClusterProtocolStackOption1
- ClusterProtocolStackOption2

### Default:

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

## ClusterProtocolStackOption1

See ClusterProtocolStack.

Optional string that can contain additional data for the cluster protocol stack. ClusterProtocolStack references this string as \${option1}, replacing \${option1} with the value of ClusterProtocolStackOption1. This string is a literal substitution of \${option1}. As such, the string must start with a ; (semicolon), the UDP parameter delimiter.

See the *JGroups* documentation for the types of values that you can assign to it. For example, set the value to the following so that a multi-home server can specify which NIC (Network Interface Card) to use for JGroups.

```
;bind_addr=xyz
```

For example, to specify specific bind addresses for individual cluster nodes that exist on multiple virtual servers, add something similar to config.xml:

```
<param value=";bind_addr=xx.xxx.xx.x" name="ClusterProtocolStackOption1" env="p1" server="prod10"/>
<param value=";bind_addr=xx.xxx.xx.x" name="ClusterProtocolStackOption1" env="p1" server="prod11"/>
<param value=";bind_addr=xx.xxx.xx.y" name="ClusterProtocolStackOption1" env="p1" server="prod12"/>
```

**Note:** To set the bind\_addr bind address property for JBoss, you must start JBoss with the system property `-Dignore.bind.address=true`. See “Specifying the Bind Address” on page 19 in the *Installation Guide*.

This configuration parameter is empty by default. It is possible to set the values of `ClusterProtocolStackOption1` and `ClusterProtocolStackOption2` by environment. Thus, different environments (development and production, for example) can use a different set of cluster protocol stack options.

**Default:** `None`

**Set for Environment:** Yes

## ClusterProtocolStackOption2

See `ClusterProtocolStack`.

Optional string that can contain additional data for the cluster protocol stack. Reference this option in the stack as  `${option2}` . This string is a literal substitution of  `${option2}` . As such, the string must start with a ; (semicolon), the UDP parameter delimiter.

See the *JGroups* documentation for the types of values that you can assign to it.

This configuration parameter is empty by default. It is possible to set the values of `ClusterProtocolStackOption1` and `ClusterProtocolStackOption2` by environment. Thus, different environments (development and production, for example) can use a different set of cluster protocol stack options.

**Default:** `None`

**Set for Environment:** Yes

## ClusterStatisticsMonitorIntervalMins

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

**Default:** 60

## ConfigVerificationEnabled

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

---

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

---

**Default:** `True`

**Set for Environment:** Yes

## JGroupsClusterChannel

Whether to use JGroups based cluster channel.

**Default:** `true`

## JGroupsWatchdogHeartbeatIntervalSecs

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

**Default:** 30

**Set for Environment:** No

## JGroupsWatchdogMissedHeartbeatsBeforeReset

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

**Default:** 10

**Set for Environment:** No

## PDFMergeHandlerLicenseKey

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

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

**Default:** None

**Set for Environment:** Yes

# Database Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## DisableHashJoinForAgencyCycleException

Guidewire provides a work-around for certain hash join-related query plan problems that occur if executing agency cycle exception queries for the **My Agency Items** page on Oracle. This parameter controls part of the work-around. The parameter has no effect on databases other than Oracle.

**Default:** True

## DisableHashJoinForPolicySearch

BillingCenter works around hash join-related query plan problems while executing certain policy searches on Oracle. This parameter controls part of the work around and is true by default. The parameter has no effect on databases other than Oracle.

See also `DisableSortMergeJoinForPolicySearch`.

**Default:** True

## DisableIndexFastFullScanForAgencyCycleException

BillingCenter provides a workaround for an index fast full scan query plan problem that occurs while executing agency cycle exception queries for the **My Agency Items** page on Oracle. This parameter controls part of the work-around. This parameter has no effect on databases other than Oracle.

**Default:** True

## DisableSortMergeJoinForAgencyCycleException

Guidewire provides a work-around for sort-merge join query plan problems that occur if executing agency cycle exception queries for the **My Agency Items** page on Oracle. This parameter controls part of the work-around if **DisableHashJoinForPolicySearch** is also set to **true**. The parameter has no effect on databases other than Oracle.

**Default:** True

## DisableSortMergeJoinForPolicySearch

BillingCenter works around sort merge join query plan problems while executing certain policy searches on Oracle. This parameter controls part of the workaround if **DisableHashJoinForPolicySearch** is also set to **true**. The parameter is **true** by default. This parameter has no effect on databases other than Oracle.

**Default:** True

## DiscardQueryPlansDuringStatsUpdateBatch

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

**Default:** False

## IdentifyQueryBuilderViaComments

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

The SQL comments are in the format:

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

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

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

**Default:** True

### See also

- “Enabling Context Comments in Queries on SQL Server” on page 179 in the *Gosu Reference Guide*

## IdentifyORMLayerViaComments

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

The SQL comments are in the format:

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

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

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

**Default:** False

**See also**

- “Enabling Context Comments in Queries on SQL Server” on page 179 in the *Gosu Reference Guide*

## MigrateToLargeIDsAndDatetime2

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

**Default:** False

# Document Creation and Document Management Parameters

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

**See also**

- “Configuring Guidewire Document Assistant” on page 147 in the *System Administration Guide*.
- For information on editing config.xml and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## AllowDocumentAssistant

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

- DisplayDocumentEditUploadButtons
- UseDocumentAssistantToDisplayDocuments

**Default:** false

## DisplayDocumentEditUploadButtons

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

**Default:** true

## DocumentAssistantJNLP

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

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

## DocumentContentDispositionMode

The Content-Disposition header setting to use any time that BillingCenter 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 BillingCenter uses to validate document template descriptor XML files. Specify this location relative to the following directory:

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

## MaximumFileUploadSize

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

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

**Default:** 20

## UseDocumentAssistantToDisplayDocuments

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

**Default:** true

# Domain Graph Parameters

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

For information on editing and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## DomainGraphKnownLinksWithIssues

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

```
relative_entity_name:foreign_key_property_name
```

---

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

---

**Default:** None

## DomainGraphKnownUnreachableTables

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

**Default:** None

## Environment Parameters

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

For information on editing and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

### AllocateInvoiceNumberOnInit

In all versions of BillingCenter, an invoice number is generated whenever an invoice is created. When creating a large number of invoices, such as during multiple simultaneous policy-preview operations, the generation of invoice numbers can slow application execution. A performance gain can be achieved by eliminating unnecessary invoice number generation and, instead, generating a number only when an invoice is committed.

To generate invoice numbers for committed invoices only, set the `AllocateInvoiceNumberOnInit` parameter to `false`. The default setting is `true`, which generates an invoice number whenever an invoice is created.

### AlwaysShowPhoneWidgetRegionCode

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

**Default:** `false`

**Set for Environment:** Yes

### CollapseFutureInvoicesUponCancellation

Whether to collapse future invoices into a single invoice any time that BillingCenter receives a cancellation billing instruction.

**Default:** `true`

### CommissionRemainderTreatment

How to distribute the remainder amount for a commission in pro rata calculations. Valid values are:

- `overweightFromFront`
- `overweightFromBack`
- `overweightFirst`
- `overweightLast`
- `writeoff`

**Default:** `writeoff`

**Note:** In previous releases, Guidewire called the `CommissionRemainderTreatment` configuration parameter `AgencyBillCommissionRemainederTreatment`.

### CreateRollupTxnsUponPolicyClosure

Upon policy closure, whether to create transactions that roll up amounts from Policy and Charge to the Account level. It is recommended that roll-up transactions not be created unless a specific use exists for them. By setting the parameter to `false`, the creation of extra transactions is avoided and the task of re-opening a closed policy becomes easier.

**Default:** `true`

## CurrentEncryptionPlugin

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

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

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

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

**Default:** None

### See also

- For information on the how to configure your database to support encryption, see “Encryption Integration Overview” on page 253 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 258 in the *Integration Guide*.
- For information on using the Plugins Registry editor, see “Using the Plugins Registry Editor” on page 109.

## DaysBeforeAccountIsConsideredInactive

Number of days to wait before deciding that this account inactive. Batch process `AccountInactivityBatchProcess` uses this value to determine whether an account is inactive.

**Default:** 60

## EnableChargeProRataTxCreation

Disable creation of `ChargeProRataTx` entities. BillingCenter normally creates `ChargeProRataTx` entities when a `Charge` is created with a `ChargePattern` subtype of `ProRataCharge`. The entities are subsequently processed by the `ChargeProRataTx` batch process to track earned premium. If you do not use BillingCenter to track earned premium, the `ChargeProRataTx` entities are not needed. When `EnableChargeProRataTxCreation` is set to `false`, `ChargeProRataTx` entities are not created.

If `EnableChargeProRataTxCreation` is set to `false`, it cannot be enabled again at a later time.

The `EnableChargeProRataTxCreation` parameter and `ChargeProRataTx` batch process are used only to track earned premium. If you do not use BillingCenter to track earned premium, set the parameter to `false`, and do not run the batch process. Any existing `ChargeProRataTx` entities not processed by the `ChargeProRataTx` batch process are ignored by BillingCenter.

**Default:** true

## EnableInternalDebugTools

Make internal debug tools available to developer.

**Default:** False

**Set for Environment:** Yes

## GosuSampleDataMethod

Method on Gosu class `DataSets.gs` that creates sample data.

**Default:** `createBillingCenterSampleData`

## IBillingCenterAPICDCENumRetries

The number of times BillingCenter retries billing instructions in the case of a `ConcurrentDataChangeException`. This integer value must be 1 or greater.

**Default:** 50

## IncludeClosedPoliciesInAccountAggregateBalances

Whether BillingCenter includes closed policies in the totals returned by the balance methods for an account.

**Default:** False

## InvoicePeriodCutoff

Specifies the `cutoff hour` parameter for invoice period determination. Guidewire defines the invoice period as the following:

`(Previous Invoice Date + cutoff hour) < invoice period <= (Current Invoice Date + cutoff hour)`

**Default:** 12:00 AM

## KeyGeneratorRangeSize

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

**Default:** 100

## MemoryUsageMonitorIntervalMins

How often the BillingCenter 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

## PASEffectiveTime

Effective time to use in the Policy Administration System (PAS). The PAS uses this value as the default time for any passed-in date.

**Default:** 12:01 AM

## PublicIDPrefix

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

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

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

**Required:** Yes

**Default:** *None*

## ResourcesMutable

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

### Studio Read-only Mode

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

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

To indicate the read-only status:

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

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

**Default:** `True`

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

## StrictDataTypes

Controls whether BillingCenter uses the pre-BillingCenter 3.0 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 BillingCenter installations that want to implement the new behavior.

#### **StrictDataTypes = true**

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

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

#### **StrictDataTypes = false**

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

- 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, BillingCenter does not permit a field with a validator format of `[0-9]{3}-[0-9]{2}-[0-9]{4}` to have the value `123-45-A123` in the interface. It is possible, however, to set a field to that value in Gosu code, for example. This enables you to bypass the validation set in the interface.
- Validates numeric range constraints in the interface, but not at the setter level. For example, Guidewire does not allow a field with a maximum value of `100` to have the value `200` in the interface. However, you can set the field to this value in Gosu rules, for example. This enables you to bypass the validation set in the interface.

**Default:** `True`

## UnrestrictedUserName

By default, ClaimCenter uses the `su` user as the user with unrestricted permissions to do anything in BillingCenter. To set the unrestricted user to a different user, set the value of the `UnrestrictedUserName` parameter to that user's login name.

**Default:** `su`

## Financial Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

### See also

- “Configuring Currencies” on page 113 in the *Globalization Guide*

## ProRataCalculationRemainderTreatment

How to distribute the remainder amount in pro rata calculations. Valid values are:

- `overweightFromFront`
- `overweightFromBack`
- `overweightFirst`
- `overweightLast`

**Default:** overweightFromFront

## Globalization Parameters

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

The globalization parameters include:

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

---

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

---

### See also

- For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## DefaultApplicationLanguage

Default display language for the application as a whole.

---

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

---

**Default:** en\_US

**Set for Environment:** Yes

**Permanent:** Yes.

### See also

- “Setting the Default Display Language” on page 35 in the *Globalization Guide*

## DefaultApplicationLocale

Default locale for regional formats in the application. You must set configuration parameter `DefaultApplicationLocale` to a typecode contained in the `Loca1Type` typelist.

---

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

---

**Default:** en\_US

**Set for Environment:** Yes

**Permanent:** Yes

**See also**

- “Setting the Default Application Locale for Regional Formats” on page 90 in the *Globalization Guide*

## DefaultApplicationCurrency

Default currency for the application. You must set configuration parameter `DefaultApplicationCurrency` to a typecode contained in the `Currency` typelist, even if you configure BillingCenter with a single currency.

Guidewire applications which share currency values must have the same `DefaultApplicationCurrency` setting in their respective `config.xml` files.

---

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

---

**Default:** usd

**Set for Environment:** Yes

**Permanent:** Yes

**See also**

- “Setting the Default Application Currency” on page 119 in the *Globalization Guide*

## DefaultRoundingMode

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

- UP
- DOWN
- CEILING
- FLOOR
- HALF\_UP
- HALF\_DOWN
- HALF\_EVEN

Guidewire strongly recommends that you use one of the following:

- HALF\_UP
- HALF\_EVEN

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

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

---

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

---

**Default:** HALF\_UP

**Permanent:** Yes

**See also**

- “Choosing a Rounding Mode” on page 119 in the *Globalization Guide*

## MulticurrencyDisplayMode

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

- `SINGLE`
- `MULTIPLE`

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

**Default:** `SINGLE`

**Permanent:** Yes

### See also

- See “Setting the Currency Display Mode” on page 120 in the *Globalization Guide*.

## DefaultCountryCode

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

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

[http://www.iso.org/iso/english\\_country\\_names\\_and\\_code\\_elements](http://www.iso.org/iso/english_country_names_and_code_elements)

## DefaultPhoneCountryCode

The default ISO country code used for phone data.

**Default:** None

## DefaultNANPACountryCode

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

**Default:** US

## AlwaysShowPhoneWidgetRegionCode

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

**Default:** false

## Integration Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## DefaultXmlExportEncryptionId

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

**Default:** null (no encryption)

## LockPrimaryEntityDuringMessageHandling

If it is set to `true`, BillingCenter 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

## PluginStartupTimeout

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

**Default:** 60

# Logging Parameters

Guidewire provides the following configuration parameters in `config.xml` that relate to application logging:

<code>LoggerCategorySource</code>	Specifies logger categories to preload at server startup.
<code>LoggersShowLog4j</code>	Determines whether loggers defined in log4j shows separately in the Set Log Level page.
<code>LoggersShowPredefined</code>	Determines whether predefined loggers show separately in the Set Log Level page.

These parameters control the selection lists of logging categories shown in the `Server Tools → Set Log Level` page. The following table lists the results of the interactions between the logging parameters.

<code>LoggerShowLog4j</code>	<code>LoggerShowPredefined</code>	<b>Result</b>
True	True	Shows drop-down picker that enables a selection list of predefined logger categories or a separate selection list of log4j logger categories that are not predefined.
True (default)	False (default)	Shows a single selection list that includes both predefined Guidewire loggers and non-categorized log4j loggers.
False	True	Shows a selection list of predefined logger categories (including any log4j logger categories that are predefined).
False	False	Disables display of any logger categories on the Set Log Level page.

### See also

- “Configuring Logging” on page 21 in the *System Administration Guide*
- “Understanding Logging Categories” on page 23 in the *System Administration Guide*

## LoggerCategorySource

A logger category represents a pre-defined Logger. Use parameter `LoggerCategorySource` to specify a class with predefined logger categories. BillingCenter pre-loads the logger categories in this class at server startup.

The specified class must one of the following logger classes (or subclass one of these classes):

- `gw.pl.logging.LoggerCategory`
- `gw.api.system.PLLoggerCategory`
- `gw.api.system.BCLoggerCategory`

The specified value must include the fully qualified class name, for example:

```
gw.pl.logging.LoggerCategory
```

**Default:** `None`

**Set For Environment:** Yes

### See also

- “Configuring Logging” on page 21 in the *System Administration Guide*
- “Understanding Logging Categories” on page 23 in the *System Administration Guide*
- “Logging Parameters” on page 51

## LoggersShowLog4j

Determines whether loggers defined in log4j show in a separate selection list in the Set Log Level page. See discussion in “Logging Parameters” on page 51.

**Default:** True

**Set For Environment:** Yes

## LoggersShowPredefined

Determines whether predefined loggers show in a separate selection list in the Set Log Level page. See discussion in “Logging Parameters” on page 51.

**Default:** False

**Set For Environment:** Yes

## Miscellaneous Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## ConsistencyCheckerThreads

Number of threads to use when running the consistency checker.

**Default:** 1

## EquityDatingIncludeWriteoffs

Whether to consider policy write-offs in equity dating calculations while calculating the policy equity and paid through date.

**Default:** False

## ListViewPageSizeDefault

The default number of entries that BillingCenter 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

## MaxTAccountsInConsistencyCheck

Maximum number of T-accounts to process in a query at one time. Set this parameter only if encountering performance issues with very large databases. See “Checking Database Consistency” on page 36 in the *System Administration Guide*.

**Default:** 0 (disabled)

## MaxTransactionsInConsistencyCheck

Maximum number of transactions to process in a query at one time. Set this parameter only if encountering performance issues with very large databases. See “Checking Database Consistency” on page 36 in the *System Administration Guide*.

**Default:** 0 (disabled)

## ProfilerDataPurgeDaysOld

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

**Default:** 30

## TransactionIdPurgeDaysOld

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

**Default:** 30

# PDF Print Settings Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## DefaultContentDispositionMode

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

**Default:** attachment

## PrintFontFamilyName

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

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

**Default:** san-serif

## PrintFontSize

Font size of standard print text.

**Default:** 10pt

## PrintFOPUserConfigFile

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

C:\fopconfig\fop.xconf

For more information, refer to the following:

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

**Default:** None

## PrintHeaderFontSize

Font size of headers in print text.

**Default:** 16pt

## PrintLineHeight

Total size of a line of print text.

**Default:** 14pt

## PrintListViewBlockSize

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

**Default:** 20

## PrintListViewFontSize

Font size of text within a list view.

**Default:** 10pt

## PrintMarginBottom

Bottom margin size of print page.

**Default:** 0.5in

## PrintMarginLeft

Left margin size of print page.

**Default:** 1in

## PrintMarginRight

Right margin size of print page.

**Default:** 1in

## PrintMarginTop

Top margin size of print page.

**Default:** 0.5in

## PrintMaxPDFInputFileSize

During PDF printing, BillingCenter 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, BillingCenter generates an error.

**Default:** -1

## PrintPageHeight

Total print height of page.

**Default:** 8.5in

## PrintPageWidth

Total print width of page.

**Default:** 11in

---

**IMPORTANT** To determine whether your Guidewire PolicyCenter license agreement includes Guidewire Rating Management, contact your Guidewire sales representative. Rating Management requires an additional license key. For instructions on obtaining and installing this key, contact your Guidewire support representative.

---

## Scheduler and Workflow Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

### SchedulerEnabled

Whether to enable the internal batch process application scheduler. See “Batch Processing” on page 107 in the *System Administration Guide* for more information on batch processes and the scheduler.

**Default:** True

**Can Change on Running Server:** Yes

### WorkflowLogDebug

Configuration parameter `WorkflowLogDebug` takes a Boolean value:

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

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

**Default:** True

### WorkflowLogPurgeDaysOld

Number of days to retain workflow log information before `PurgeWorkflowLogs` batch processing deletes it.

See “Purge Workflow Logs Batch Processing” on page 139 in the *System Administration Guide* for details.

**Default:** 30

### WorkflowPurgeDaysOld

Number of days to retain workflow information before `PurgeWorkflows` batch processing deletes it.

See “Purge Workflow Batch Processing” on page 139 in the *System Administration Guide* for details.

**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 disables statistics reporting.

**Default:** 60

## Search Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## MaxContactSearchResults

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

**Default:** 100

## MaxSearchResults

Maximum number of results that BillingCenter search screens will return. If the number of search results is greater than this value, BillingCenter prompts the user to narrow the search parameters. To specify that the number of search results is not limited, set this parameter to -1.

You can also override the `MaxSearchResults` parameter for individual search screens. Each search screen that uses the `MaxSearchResults` configuration parameter has a corresponding `maxSearchResults` PCF property. To set a different search result limit for a particular screen, edit the `maxSearchResults` PCF property for that screen.

The contact search screen and the payment search screen do not use the `MaxSearchResults` configuration parameter. The contact search screen uses `MaxContactSearchResults`. The payment search screen includes multiple result tabs and does not limit the number of search results.

**Default:** 300

### See also

- “Configuring BillingCenter Search” on page 316

# Security Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## 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 BillingCenter 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 BillingCenter 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 0 or greater.

**Default:** 0

**Minimum:** 0

## MaxPasswordLength

New passwords must be no more than this many characters long. This integer value must be 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 0 or greater. If 0, then Guidewire BillingCenter does not require a password. (Guidewire does not recommend this.)

**Default:** 8

**Minimum:** 0

## RestrictContactPotentialMatchToPermittedItems

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

**Default:** True

## RestrictSearchesToPermittedItems

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

**Default:** True

## SessionTimeoutSecs

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

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

This value sets the session expiration timeout globally for all BillingCenter browser sessions. See “Configuring Client Session Timeout” on page 62 in the *System Administration Guide* for information on how to set the session timeout at a more granular level.

**Default:** 10800

**Minimum:** 300

**Maximum:** 604800

## User Interface Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

### ActionsShortcut

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

**Default:** A

### AutoCompleteLimit

The maximum number of autocomplete suggestions to show.

**Default:** 10

### InputMaskPlaceholderCharacter

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

**Default:** . (period)

### ListViewPageSizeDefault

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

**Default:** 15

**Minimum:** 1

### MaxBrowserHistoryItems (Obsolete)

**This parameter is obsolete. Do not use it.**

### QuickJumpShortcut

The keyboard shortcut to use to activate the QuickJump box.

**Default:** / (forward slash)

### UISkin

Name of the BillingCenter interface skin to use.

**Default:** Titanium

### WizardNextShortcut

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

## WizardPrevShortcut

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

## WizardPrevNextButtonsVisible

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

**Default:** `False`

# Work Queue Parameters

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

For information on editing `config.xml` and setting configuration parameters, see “Working with Configuration Parameters” on page 28.

## EnableWorkQueueValidation

Specifies whether to perform bundle validation on BillingCenter-specific work queues.

Bundle validation helps to preserve data consistency when entities are updated. This goal is achieved by defining and running validation rules on every commit.

Validation rules are not supported for all entities. For the list of entities that can trigger validation rules, see “Validation” on page 40 in the *Rules Guide*.

This parameter affects BillingCenter-specific work queues only. Guidewire platform work queues, such as `Activity Escalation`, `Group/User Exception`, and `Contact Auto Sync`, always perform validation, regardless of the parameter setting.

Validation can significantly impact performance. For that reason, enabling validation is not recommended unless the situation requires it.

The BillingCenter base configuration does not define any validation rules. Accordingly, the default setting of the `EnableWorkQueueValidation` parameter is `False`.

**Default:** `False`

## InstrumentedWorkerInfoPurgeDaysOld

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

**Default:** 45

## WorkItemCreationBatchSize

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

**Default:** 100

## WorkItemPriorityMultiplierSecs

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

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

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

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

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

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

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

---

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

---

**Default:** 600

## WorkItemRetryLimit

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

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

The value for `WorkItemRetryLimit` applies to all work queues unless overridden in `work-queue.xml` by `retryLimit` for specific work queues. For more information on tuning work queue performance by adjusting the number of retries, see “Configuring Work Queues” on page 116 in the *System Administration Guide*.

**Default:** 3

## WorkQueueHistoryMaxDownload

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

**Default:** 10000

## WorkQueueThreadPoolMaxSize

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

**Default:** 50

**Set For Environment:** Yes

## WorkQueueThreadPoolMinSize

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

**Default:** 0

**Set For Environment:** Yes

## WorkQueueThreadsKeepAliveTime

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

**Default:** 60

**Set For Environment:** Yes

# The Guidewire Development Environment



# Getting Started

This topic describes Guidewire Studio, which is the BillingCenter administration tool for creating and managing BillingCenter resources. (Studio resources include Gosu rules, classes, enhancements, script parameters, and the BillingCenter 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:

- “What Is Guidewire Studio?” on page 65
- “The Studio Development Environment” on page 66
- “Working with the QuickStart Development Server” on page 67
- “BillingCenter Configuration Files” on page 69
- “Using Studio with IntelliJ IDEA Ultimate Edition” on page 70
- “Studio and the DCE VM” on page 70
- “Starting Guidewire Studio” on page 71
- “Using the Studio Interface” on page 72

## What Is Guidewire Studio?

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

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

Using Guidewire Studio, you can:

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

---

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

---

## The Studio Development Environment

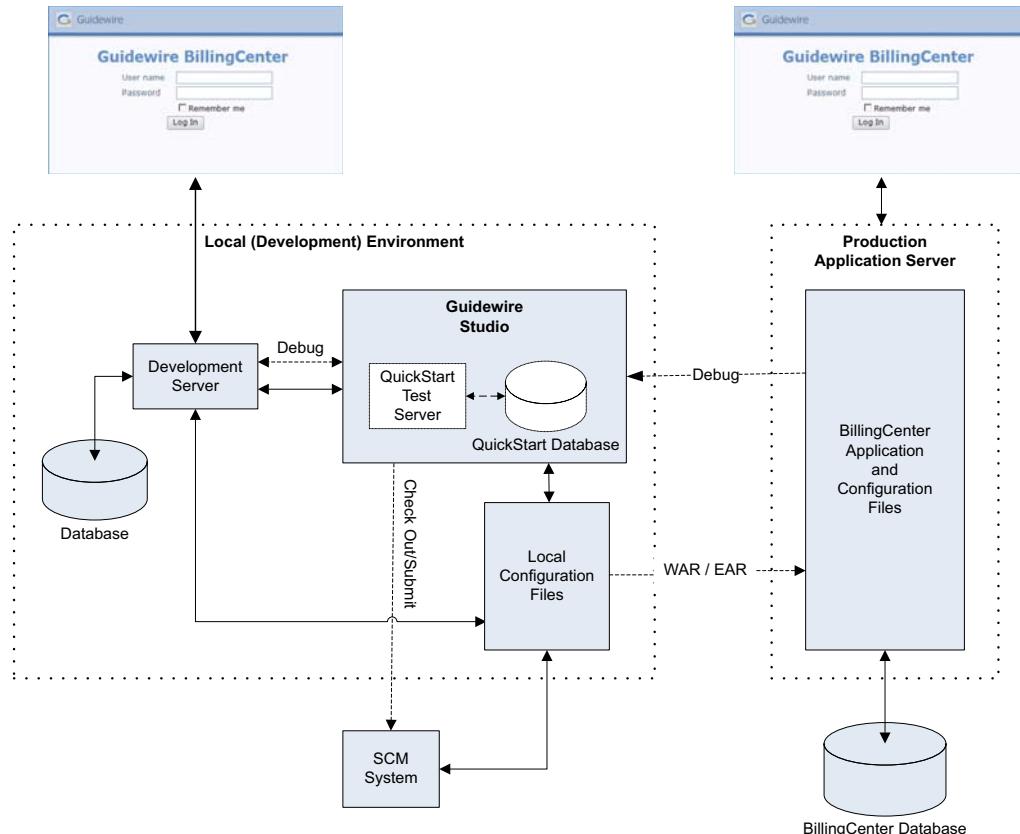
Guidewire Studio is a stand-alone development application that runs independently of Guidewire BillingCenter. You use Studio to build and test application customization in a development or test mode before deploying your changes to a production server. Any changes that you make to application files through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

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

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



## Working with the QuickStart Development Server

It is possible to use any of the supported application servers in a development environment, rather than the embedded QuickStart server. To do so, you need to point BillingCenter to the configuration resources edited by Guidewire Studio. This requires additional configuration, described for each application server type in “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.

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

```
BillingCenter/bin/gwbc dev-start
```

Use the following `dev` commands as you work with the QuickStart server. See “Commands Reference” on page 97 in the *Installation Guide* for a complete list of commands and how to use them.

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

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

Application	Port
ClaimCenter	8080
PolicyCenter	8180
ContactManager	8280
BillingCenter	8580

For more information on the gwbc dev commands, see “Installing the QuickStart Development Environment” on page 46 in the *Installation Guide*.

## Connecting the Development Server to a Database

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

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

### To drop the QuickStart database

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

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

## Deploying Your Configuration Changes

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

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

### To hot-deploy PCF files

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

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

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

## BillingCenter Configuration Files

---

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

---

Installing Guidewire BillingCenter creates the following directory structure:

Directory	Description
.idea	Contains configuration and settings for IntelliJ IDEA.
admin	Contains administrative tools. See “Using BillingCenter Command Prompt Tools” on page 185 in the <i>System Administration Guide</i> for descriptions.
bin	Contains the gwbc batch file and shell script used to launch commands for building and deploying. See “Commands Reference” on page 97 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 BillingCenter. 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 BillingCenter documentation.
idea	Contains IntelliJ IDEA application files.
java-api	Contains the Java API libraries created by running the gwbc regen-java-api command. See “Regenerating Integration Libraries and WSDL” on page 20 in the <i>Integration Guide</i> .
logs	Contains log files.
modules	Contains subdirectories including configuration resources for each application component.
repository	Contains necessary BillingCenter files.
soap-api	Contains the web service WSDL files that the gwbc regen-soap-api command generates. See “Regenerating Integration Libraries and WSDL” on page 20 in the <i>Integration Guide</i> .
solr	For internal use only.
studio	Contains Studio preferences and TypeInfo database caches. Studio generates this directory when you first launch Studio.

---

template	Contains template files.
webapps	Contains necessary files for use by the application server.

---

#### **Edited Resource Files Reside in the Configuration Module Only**

The configuration module is the only place for configured resources. As BillingCenter 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, BillingCenter does not start. In this case, overwrite any changes to all modules except for the configuration directory and try again.

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

## Key Directories

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

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

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

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

## Using Studio with IntelliJ IDEA Ultimate Edition

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

#### **To configure Guidewire Studio to use IntelliJ IDEA Ultimate Edition**

1. In your BillingCenter installation directory, create a text file named `studio.ultimate` that contains the full path of your IntelliJ IDEA Ultimate Edition installation directory. For example:  
`C:\Program Files (x86)\JetBrains\IntelliJ IDEA 12.1.7`
2. Run Guidewire Studio.
3. When prompted for your IntelliJ IDEA Ultimate Edition license, provide it.

## Studio and the DCE VM

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

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

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

### DCE VM Limitations

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

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

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

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

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

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

```
MyClass.x = 10
```

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

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

### See also

- For details on how to select the proper JVM for your installation, see “Installing Java” on page 40 in the *Installation Guide*.
- “Installing the Dynamic Code Evolution Virtual Machine” on page 40 in the *Installation Guide*.

## Starting Guidewire Studio

You start Studio from the command line.

### To start Guidewire Studio

4. Do either of the following:

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

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

### To stop Guidewire Studio

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

## Restarting Studio

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

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

**Note:** If you modify the base configuration data model, you must start (or restart) the application server. This forces a database upgrade. See “Deploying Configuration Files” on page 23 for more information.

## Using the Studio Interface

The Guidewire Studio interface consists of either two or more large panes (depending on which resource you select) arranged from left-to-right.

Page element	Description
Left-most pane	The left-most or 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: <ul style="list-style-type: none"> <li>Green indicates that the Gosu code is valid.</li> <li>Yellow indicates that the Gosu code is valid but contains warnings.</li> <li>Red indicates that the Gosu code is invalid. This code does not compile.</li> </ul>

### Resizing Studio Panes

It is possible to resize a Studio pane by grabbing the edge of the pane and dragging. To alert you to this possibility, Studio displays a *splitter* icon at the edge of the pane. It looks like a short row of multiple dotted lines, like this:



To re-size a window pane, simply drag the edge of the pane into the new position.

### 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 BillingCenter 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**



# 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 75
- “Accessing Reference Information” on page 79
- “Using Studio Keyboard Shortcuts” on page 80
- “Viewing Keyboard Shortcuts in BillingCenter” on page 86
- “Using Text Editing Commands” on page 87
- “Navigating Tables” on page 87
- “Refactoring Gosu Code” on page 88
- “Saving Your Work” on page 89
- “Validating Studio Resources” on page 89

## Entering Valid Code

Guidewire Studio provides several different ways of obtaining information about BillingCenter 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 works with both Gosu or Java packages.
SmartHelp	Displays a list of valid fields and subobjects for the current object.

Keyboard commands provide code completion, code navigation, and code editing shortcuts. See “Using Studio Keyboard Shortcuts” on page 80 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	<p>Opens the <b>Go To Type</b> 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"> <li>Simple wildcard searches with multiple asterisks in the search string</li> <li>Camel-case searching in which you type part or all of an acronym. For example, entering CrA returns a list including <code>CreateActivityPattern</code> and <code>CreateAttribute</code>.</li> </ul>
Go to Symbol	ALT+CTRL+SHIFT+N	Opens the <b>Go to Symbol</b> 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 <b>Go To Line</b> 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	<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 <code>Sides</code> in the <code>Triangle</code> class and pressing CTRL+B takes you to the corresponding declaration in class <code>Shape</code>.</p> <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+slash	<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+Slash to open the object value selection dialog. For example, typing the following:</p> <pre>Activity.AssignedByUser ==</pre> <p>and then pressing CTRL+Slash, opens a selection dialog in which you can choose a specific user from the User list.</p>

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"><li>• Completing a partial word.</li><li>• Completing a property or method on a partially formed expression.</li><li>• Automatically adding a “uses” statement for a type that has not been imported yet.</li></ul> 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 <b>Objects and Functions</b> 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.
Complete Class Name	ALT+CTRL+SPACE	Opens a list of resources starting with the initial letters that you typed.
Parameter Info	CTRL+P	Displays information on the parameter at the current point in a method call.

## Using Dot Completion

You can use Studio to complete your code by placing the cursor (or caret) at the end of a valid BillingCenter 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 tries to assist 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"><li>• 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.</li><li>• Search... opens a dialog box in which you can enter criteria to search for the correct entity.</li></ul>

## Accessing Reference Information

BillingCenter provides reference information that you access from the Studio Help menu. This includes:

Gosu API Reference	Provides a searchable reference on the Gosu APIs, methods and properties. See “Gosu Generated Documentation (Gosudoc)” on page 38 in the <i>Gosu Reference Guide</i> .
PCF Reference Guide	Provides a description of the PCF widgets and their attributes that you can use within the PCF editor. This documentation is also known as the <i>PCF Format Reference</i> .
Gosu Reference Guide	Provides information on the Gosu language. You can access the entire Guidewire documentation suite from this menu link.

### Accessing the Gosu API Reference

Guidewire provides API reference information 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:

- Access the Gosu API reference as described at “Gosu Generated Documentation (Gosudoc)” on page 38 in the *Gosu Reference Guide*. Use the Search functionality to find information on an API, or expand the API list and select a field, parameter or method to view.
- Click a method name in a line of code and press CTRL+Q. A pop-up window opens and displays information about that method, including information about its parameters.

The Gosu API reference window contains a search pane, a contents tree, and a display area.

- The contents tree displays a tree view of the type system, organized by package.
- The search pane 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 index it before proceeding.

### Accessing the PCF Reference Guide

Guidewire provides a PCF reference (official name, *Guidewire BillingCenter 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 BillingCenter 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 BillingCenter 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"><li>• Intelligent Coding Commands</li><li>• Code Navigation Commands</li><li>• Refactoring Commands</li><li>• Code Editing Commands</li><li>• Find Commands</li><li>• Debugging Commands</li><li>• Help Commands</li><li>• Testing Commands</li><li>• General Commands</li></ul>
Gosu Tester	<ul style="list-style-type: none"><li>• Gosu Tester Commands</li></ul>
PCF Editor	<ul style="list-style-type: none"><li>• PCF Editing Commands</li></ul>

### Gosu Editor

The following keystroke shortcuts work in the Studio Gosu editor (within rules and classes, for example).

- Intelligent Coding Commands
- Code Navigation Commands
- Refactoring Commands
- Code Editing Commands
- Find Commands
- Debugging Commands
- Help Commands
- Testing Commands
- General Commands

## 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"> <li>Completing a partial word.</li> <li>Completing a property or method on a partially formed expression.</li> <li>Automatically adding a “uses” statement for a type that has not been imported yet.</li> </ul> <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 <b>Objects and Functions</b> 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+/ Space	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+/ Space to open the object value selection dialog. For example, typing the following:</p> <pre>Activity.AssignedByUser ==</pre> <p>and then pressing CTRL+/ Space, opens a selection 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.

## Code Navigation Commands

Key	Name	Description
CTRL+N	Go To Type	<p>Opens the <b>Go To Type</b> popup that enables quick navigation to other types.</p> <p>For example, enter a search string to find a Gosu class. You can perform:</p> <ul style="list-style-type: none"> <li>Simple wild-card searches with multiple asterisks in the search string.</li> <li>Camel-case searches in which you type part or all of an acronym. For example, entering CrA returns a list including CreateActivityPattern and CreateAttribute.</li> </ul> <p>Selecting a type from the list opens the editor view for that type. You can also depress the keyboard <b>Enter</b> key to select the first type in the list.</p> <p>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"> <li>Entering GL*Enh finds GLCostSetEnhancement. It does not find GeneralLiabilityLineEnhancement.</li> <li>Entering GLL does find GeneralLiabilityLineEnhancement.</li> </ul>
CTRL+ALT+SHIFT+N	Go To Symbol	<p>Opens the <b>Go to Symbol</b> popup, which you can use to quickly navigate to symbols (class fields, PCF variables, and similar items) in all types.</p> <p>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 <b>Recent Views</b> popup, which you can use to quickly navigate to recently edited files.
CTRL+G	Go To Line	Opens the <b>Go To Line</b> popup, which you can use to quickly navigate to particular lines in a file.

Key	Name	Description
CTRL+B	Go To Declaration	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     functon Shape() { } }</pre>
CTRL+SHIFT+B	Got to Enhancement	Opens a secondary menu from which you can create a new enhancement.
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.
CTRL+L	Center View	Centers the line of the current caret location within the editor view.
HOME	Beginning of Line	Move caret to beginning of the current line. If the caret is at the beginning of the line, move the caret to the end of the opening indentation.
END	End of Line	Move the caret to the end of the current line.
CTRL+HOME	Beginning of File	Move the caret to the beginning of the current file.
CTRL+END	End of File	Move the caret to the end of the current file.
CTRL+LEFT ARROW	Previous Word	Move the caret to the beginning of the previous word.
CTRL+RIGHT ARROW	Next Word	Move the caret to the end of the previous word.
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.

### See also

- “Refactoring Gosu Code” on page 88

## Code Editing Commands

Key	Name	Description
CTRL+DELETE	Delete Word	Deletes the word after the editor caret.
CTRL+BACKSPACE	Delete Word Before	Deletes the word before the editor caret.
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+V	Paste	Paste the contents of the copy buffer at the current caret location.
CTRL+SHIFT+V	Show Copy Buffer	Shows the copy buffer dialog, allowing a previous cut or copy to be selected for paste.
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.
CTRL+SHIFT+LEFT ARROW	Select Word Previous	Expand or narrow the current selection to include or exclude the word before the current caret location.
CTRL+SHIFT+RIGHT ARROW	Select Word Next	Expand or narrow the current selection to include or exclude the word after the current caret location.

## Find Commands

Key	Name	Description
CTRL+F	Find	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:  <code>var test = 5</code>
CTRL+R	Replace	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.
F3	Find Next	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.
SHIFT+F3	Find Previous	Performs the previous search in the opposite direction. The search algorithm does not repeat a match until it cycles through all other matches in the open view.

Key	Name	Description
ESCAPE		Cancels the current search and removes the highlight.
ALT+F7	Find Usages	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.
CTRL+SHIFT+F	Find in Path	Searches for a resource within Studio using the provided resource name or text string.
CTRL+SHIFT+R	Replace in Path	Searches for a text string and replace its value with the supplied value.
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"> <li>• Right-click the active tab and select <b>Find in Resources</b>.</li> <li>• Select <b>Find Selected View in Resources</b> from the toolbar <b>Window</b> menu.</li> </ul>

## Debugging Commands

Key	Name	Description
F9	Resume	Resume code execution
CTRL+F2	Stop	Stop debugging.
ALT+F10	Show Execution Point	Open (focus) the editor view at the current execution point.
F7	Step Into	Execute the next instruction at the current execution point. If it is a method call, stop at the first line of the method.
F8	Step Over	Execute the line of code at the current execution point.
CTRL+F8	Toggle Breakpoint	Sets or removes a breakpoint at the current caret location. You cannot set breakpoints on lines that do not correspond to executable code (blank lines, comments, for example).

### See also

- “Testing and Debugging Your Configuration” on page 383

## 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.

## Testing Commands

Key	Name	Description
CTRL+F10	Run Current	Run the current configuration.
CTRL+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.

**See also**

- “Using GUnit” on page 391

**General Commands**

Key	Name	Description
CTRL+S	Save Changes	Save any unsaved changes.
CTRL+H	Refresh	Refreshes the view that Studio has of all files. Use at the appropriate time if you have updated files outside of Studio and you disabled the File Refresh option.
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

**See also**

- “Setting Font Display Options” on page 93
- “Saving Your Work” on page 89

**Gosu Tester**

The following keystroke shortcuts work in the Studio Gosu tester.

**Gosu Tester Commands**

Key	Name	Description
CTRL+Z	Undo	Undo the last action.
CTRL+Y or CTRL+ALT+F12	Redo	Redo the last action that you undid.
F5 or CTRL+ENTER	Run	Run the code in the Tester editor.
F2 or ALT+X	Clear and Run	Clears the Tester window and runs the code in the Tester editor.

**See also**

- “Using the Gosu Scratchpad” on page 389

**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.

Key	Name	Description
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 comment out a widget.)

**See also**

- “Using the PCF Editor” on page 269

## Viewing Keyboard Shortcuts in BillingCenter

It is possible to view a list of the shortcut keys that are specific to a BillingCenter 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 BillingCenter 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	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.
Edit → Redo		The undo/redo functionality is context sensitive. For example: <ul style="list-style-type: none"> <li>If you select the Rules tab, undo/redo operates on rule operations like adding or removing a rule, and similar operations.</li> <li>If you select a specific rule, undo/redo operates on operations within that rule only.</li> <li>If you select a class editor, undo/redo operates exclusively on that editor. This does not affect any changes made in the Rules tab.</li> </ul> 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 as expected. 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 similar manner 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. BillingCenter 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.)

**Note:** 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.

- BillingCenter 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.
- BillingCenter 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

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

- If you move between different tabs (views) within Studio
- If the Studio main window loses focus (for example, by moving to another application)

However, you also have the option of performing manual “saves” 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).

## 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.

For the verify commands to work correctly, you must choose an appropriate root resource. For example, to verify an entity extension, select the **Data Model Extensions** node in the **Resource** pane, then one of the verify options. As the data model type definitions depend on each other, you must validate all of them together.

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.

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

### Verifying PCF Files

The process of verifying PCF files can be time consuming, especially if the PCF files are modal and contain common sections. To improve performance, Guidewire provides an option to limit the number of second-pass verifications that take place during PCF verification.

# Working with Guidewire Studio

This topic describes Guidewire Studio and the Studio development environment.

This topic includes:

- “Improving Studio Performance” on page 91
- “Setting Font Display Options” on page 93
- “Setting Font Display Options” on page 93

## Improving Studio Performance

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

- You can make code compile faster or not use up Studio memory. See “Improving Performance of Code Compilation” on page 91.
- You can provide overall improvement by providing Studio with more memory. See “Increasing Memory Available to Studio” on page 92.

### Improving Performance of Code Compilation

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

#### Gosu Hints for Improving Gosu Compilation Speed

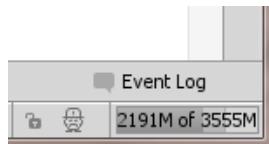
There are several things you can do with Gosu to improve compile speed:

- Write all your Gosu as case-sensitive code. See “Gosu Case Sensitivity” on page 96.
- Configure Gosu class preloading. See “Preloading Gosu Classes” on page 100.

## Using the External Compiler

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

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



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

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

### To set Studio to use the external compiler

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

## Increasing Memory Available to Studio

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

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

- **Starting heap size (.xms)** – 2000 megabytes
- **Maximum heap size (.xmx)** – 4000 megabytes
- **Maximum permanent size (.maxperm)** – 500 megabytes

### To change the Studio memory settings

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

## Setting Font Display Options

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

Using this menu, you can set values for:

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

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

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

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

### Namespace URLs

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

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

### Configuring External XML Editors

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

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



# BillingCenter Studio and Gosu

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

This topic includes:

- “Gosu Building Blocks” on page 95
- “Gosu Case Sensitivity” on page 96
- “Working with Gosu in BillingCenter Studio” on page 96
- “Gosu Packages” on page 97
- “Gosu Classes” on page 97
- “Gosu Enhancements” on page 100
- “The Guidewire XML Model” on page 101
- “Script Parameters” on page 101

## Gosu Building Blocks

Guidewire provides a number of building blocks to assist you in implementing, configuring, and testing your business logic in BillingCenter. 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 197 in the *Gosu Reference Guide*.
- For information on the BillingCenter base configuration classes see, “BillingCenter Base Configuration Classes” on page 98.

- For information on the `@export` annotation and how it affects a class in Studio, see “Class Visibility in Studio” on page 99.
- For general information on Gosu enhancements, see “Enhancements” on page 235 in the *Gosu Reference Guide*.
- For information on using Gosu business rules within Guidewire BillingCenter, see “Rules Overview” on page 15 in the *Rules Guide*.
- For information on script parameters and how to use them in Gosu code, see “Script Parameters” on page 101.

## Gosu Case Sensitivity

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

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

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

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

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

## Working with Gosu in BillingCenter Studio

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

Classes → New →	For information, see...
<b>Class</b>	• “Classes” on page 197 in the <i>Gosu Reference Guide</i> • “Gosu Classes” on page 97
<b>Interface</b>	• “Interfaces” on page 219 in the <i>Gosu Reference Guide</i>
<b>Enhancement</b>	• “Enhancements” on page 235 in the <i>Gosu Reference Guide</i> • “Gosu Enhancements” on page 100
<b>Template</b>	• “Gosu Templates” on page 359 in the <i>Gosu Reference Guide</i>

Classes → New →	<b>For information, see...</b>
Package	• “Gosu Packages” on page 97
GX Model	• “The Guidewire XML (GX) Modeler” on page 310 in the <i>Gosu Reference Guide</i>

## Gosu Packages

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

### To create a new package

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

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

**Note:** You can only delete an empty package.

## Gosu Classes

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

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

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

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

### To create a new class

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

### See also

- “BillingCenter Base Configuration Classes” on page 98
- “Class Visibility in Studio” on page 99
- “Preloading Gosu Classes” on page 100
- “Classes” on page 197 in the *Gosu Reference Guide*

## BillingCenter Base Configuration Classes

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

- **com**
- **gw**
- **libraries**
- **wsi**

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

### The com Package

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

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

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

### The gw Package

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

- **gw.api.\***
- **gw.plugin**

#### **gw.api.\***

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

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

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

- **activity**
- **address**
- **admin**
- **...**

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

#### **gw.plugin**

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

- For information on how to use Studio to work with plugins, see “[Using the Plugins Registry Editor](#)” on page 109.
- For information on various types of plugins and how to implement plugins, see “[Plugin Overview](#)” on page 135 in the *Integration Guide*.

## The libraries Package

The `libraries` package contains a number of pre-built extensions that provide additional functionality. To access these functions, enter the full package name (for example):

```
libraries.AccountExt.getDelinquencyReasons()
```

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.AccountExt  
...  
AccountExt.getDelinquencyReasons()
```

## The wsi Package

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

### See also

- “Web Services Introduction” on page 27 in the *Integration Guide*.
- “Reference of All Built-in Web Services” on page 29 in the *Integration Guide*.
- “Calling Web Services from Gosu” on page 65 in the *Integration Guide*.
- “Using the Web Service Editor” on page 116.

## Class Visibility in Studio

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

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

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

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

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

You then have this class extend `CCPCSearchCriteria`, for example:

```
package gw.webservice.pc.ccintegration.v2.ccentities  
  
uses java.util.Date  
uses gw.api.web.product.ProducerCodePickerUtil  
uses gw.api.web.producer.ProducerUtil  
  
class MyClass extends CCPCSearchCriteria {  
    var _accountNumber : String as AccountNumber  
    var _asOfDate : Date as AsOfDate  
    var _nonRenewalCode : String as NonRenewalCode  
    var _policyNumber : String as PolicyNumber  
    var _policyStatus : String as PolicyStatus  
    var _producerCodeString : String as ProducerCodeString  
    var _producerString : String as ProducerString  
    var _product : String as Product  
    var _productCode : String as ProductCode  
    var _state : String as State  
    var _firstName : String as FirstName  
    var _lastName : String as LastName  
    var _companyName : String as CompanyName  
    var _taxID : String as TaxID  
  
    construct() { }  
  
    override function extractInternalCriteria() : PolicySearchCriteria {  
        var criteria = new PolicySearchCriteria()  
        ...  
    }  
}
```

```

        criteria.SearchObjectType = SearchObjectType.TC_POLICY
    ...
}

```

## Preloading Gosu Classes

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

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

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

Guidewire provides the logging category `Server.Preload`, which provides `DEBUG` level logging of all actions and preloaded types.

### Populating the List of Types

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

## Gosu Enhancements

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

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

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

The Gosu language defines the following terms:

- **Classes** – Gosu classes encapsulate data and code for a specific purpose. You can subclass and extend existing classes. You can store and access data and methods on an instance of the class or on the class itself. Gosu classes can also implement Gosu interfaces.

- **Enhancements** – Gosu enhancements are a Gosu language feature that allows you to augment classes and other types with additional concrete methods and properties. For example, use enhancements to define additional utility methods on a Java class or interface that you cannot directly modify. Also, you can use an enhancement to extend Gosu classes, Guidewire entities, or Java classes with additional behaviors.

#### To create a new enhancement

1. First create a package for your new class, if you have not already done so.
2. Select the package in the configuration tree.
3. Right-click, select **New**, then **Enhancement** from the menu.
4. Enter the name for this enhancement. Guidewire recommends strongly that you end each enhancement name with the word **Enhancement**. For example, if you create an enhancement for an **Activity** entity, name your enhancement **ActivityEnhancement**.
5. Enter the entity type to enhance. For example, if enhancing an **Activity** entity, enter **Activity**.
6. Click **OK** to save your work and exit this dialog.

#### See also

- “[Classes](#)” on page 197 in the *Gosu Reference Guide*
- “[Enhancements](#)” on page 235 in the *Gosu Reference Guide*

## The Guidewire XML Model

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

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

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

#### See also

- For detailed information on the Guidewire XML model and the Guidewire XML Modeler in Studio, see “[The Guidewire XML \(GX\) Modeler](#)” on page 310 in the *Gosu Reference Guide*.

## Script Parameters

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

This topic includes:

- “[Script Parameters Overview](#)” on page 102
- “[Working with Script Parameters](#)” on page 102
- “[Referencing a Script Parameter in Gosu](#)” on page 103

## Script Parameters Overview

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

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

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

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

---

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

---

### Script Parameters as Global Variables

There are several reasons to create global variables:

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

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

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

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

### Script Parameter Examples

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

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

## Working with Script Parameters

In working with script parameters:

- You create script parameters and set their initial values in Guidewire Studio.

- You administer script parameters and modify their values in the BillingCenter interface, on the **Administration** tab.

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

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

#### To create a script parameter

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

#### To delete a script parameter

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

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

## Referencing a Script Parameter in Gosu

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

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

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

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

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

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



---

part III

# Guidewire Studio Editors



# Using the Studio Editors

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

This topic includes:

- “Editing in Guidewire Studio” on page 107
- “Working in the Gosu Editor” on page 108

## Editing in Guidewire Studio

Guidewire Studio displays BillingCenter 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 135
Entity Names	Represent an entity name as a text string suitable for viewing in the BillingCenter interface.	“Using the Entity Names Editor” on page 125
Gosu	Create and manage Gosu code used in classes, tests, enhancements, and interfaces.	“Working in the Gosu Editor” on page 108
Messaging	Work with messaging plugins.	“Using the Messaging Editor” on page 131
Page Configuration (PCF)	Graphically define and edit page configuration (PCF) files, used to render the BillingCenter Web interface.	“Using the PCF Editor” on page 269
Plugins	Graphically define, edit and manage Java and Gosu plugins.	“Using the Plugins Registry Editor” on page 109
TypeList	Define typelists for use in the application.	“Working with TypeLists” on page 245
Workflows	Graphically define and edit application workflows.	“Using the Workflow Editor” on page 335

## Working in the Gosu Editor

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

- Classes
- Enhancements
- Interfaces
- GUnit tests

**See also**

- “Classes” on page 197 in the *Gosu Reference Guide*



## chapter 8

# Using the Plugins Registry Editor

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

This topic includes:

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

## What Are Plugins?

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

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

## Plugin Implementation Classes

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

### See also

- “Plugin Overview” on page 135 in the *Integration Guide*.

## What is the Plugins Registry?

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

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

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

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

#### See also

- For the maximum supported implementations for each plugin interface, see the table “Summary of All BillingCenter Plugins” on page 153 in the *Integration Guide*.

## Startable Plugins

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

#### See also

- “Startable Plugins Overview” on page 273 in the *Integration Guide*
- “Multi-threaded Inbound Integration Overview” on page 281 in the *Integration Guide*

# Working with Plugins

## Creating a Plugins Registry Item

### To create a plugin item

1. In the Project window, navigate to `configuration` → `config` → `Plugins` → `registry`.
2. Right-click `registry`, and then click `New` → `Plugin`.
3. In the `Name` text box, type the plugin name. If the interface supports only a single implementation, use the same name as the plugin interface. For the maximum supported implementations for each interface, see the table “Summary of All BillingCenter Plugins” on page 153 in the *Integration Guide*.
4. In the `Interface` box, type the name of the plugin interface, or click `Browse`  to search for valid interfaces. For all startable plugins, enter `IStartablePlugin`.

## Adding an Implementation to a Plugins Registry Item

### To add a plugin implementation to a plugin item

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

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

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

### Gosu Implementations

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

<b>Gosu Class</b>	Enter the name of the Gosu class that implements this plugin interface. In the base configuration, Guidewire places all Gosu plugin implementation classes in the following package in the <b>Classes</b> folder:  <code>gw.plugin.package.impl</code> You must enter the fully-qualified path to the Gosu implementation class. For example, use <code>gw.plugin.email.impl.EmailMessageTransportPlugin</code> for the Gosu EmailMessageTransport plugin. See "Example Gosu Plugin" on page 141 in the <i>Integration Guide</i> for more information.
-------------------	--

### Java Implementations

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

<b>Java Class</b>	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>BillingCenter/modules/configuration/plugins/</code>
<b>Plugin Directory</b>	(Optional) Enter the name of the base plugin directory for the Java class. This is a folder (directory) in the <code>modules/configuration/plugins</code> directory. If you do not specify a value, Studio assumes that the class exists in the <code>modules/configuration/plugins/shared</code> directory. See "Special Notes For Java Plugins" on page 142 in the <i>Integration Guide</i> .

### OSGi Implementations

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

<b>Service PID</b>	Enter the fully-qualified Java class name for your OSGi implementation class. See "Overview of Java and OSGi Support" on page 451 in the <i>Integration Guide</i> .
--------------------	--

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

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

## Enabling and Disabling a Plugin Implementation

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

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

## Setting Environment and Server Context for Plugin Implementations

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

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

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

### See also

- “Reading System Properties in Plugins” on page 151 in the *Integration Guide*
- “Specifying Environment Properties in the <registry> Element” on page 15 in the *System Administration Guide*

## Customizing Plugin Functionality

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

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

For information on plugins in general, see “Plugin Overview” on page 135 in the *Integration Guide*.

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

Plugin type	Description	See
Gosu	A Gosu class	• “Example Gosu Plugin” on page 141 in the <i>Integration Guide</i>
Java	A Java class that does not use the OSGi standard.	• “Special Notes For Java Plugins” on page 142 in the <i>Integration Guide</i> • “Overview of Java and OSGi Support” on page 451 in the <i>Integration Guide</i>
OSGi	A Java class inside an OSGi bundle.	• “Overview of Java and OSGi Support” on page 451 in the <i>Integration Guide</i>

## Working with Plugin Versions

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

For the package name, Guidewire includes the application two-digit abbreviation followed by the application version number with no periods. For BillingCenter 8.0.0, for example, the package includes bc800.

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

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

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

Guidewire uses the following abbreviation conventions for naming its applications:

Application	Abbreviation
ClaimCenter	cc
PolicyCenter	pc
ContactManager	ab
BillingCenter	bc



# Working with Web Services

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

This topic includes:

- “Web Services and Guidewire Studio” on page 115

## Web Services and Guidewire Studio

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

### To create a WS-I web service in Studio

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

BillingCenter supports WS-I web services. WS-I web services use the SOAP protocol and are compatible with the WS-I standard. For an overview of web services, including a list of all built-in web services, see “Web Services Introduction” on page 27 in the *Integration Guide*.

Topic	See
Overview of web services	“Web Services Introduction” on page 27 in the <i>Integration Guide</i>
Reference of all built-in web services	“Reference of All Built-in Web Services” on page 29 in the <i>Integration Guide</i>
Publishing a web service on the BillingCenter server	“Publishing Web Services” on page 31 in the <i>Integration Guide</i>
Consuming a web service	“Calling Web Services from Gosu” on page 65 in the <i>Integration Guide</i>

## Using the Web Service Editor

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

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

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

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

Area or item	Description
Resources	Displays the resource URLs that you have defined. Each URL points to a web service WSDL file. The WSDL file can exist in any of the following locations: <ul style="list-style-type: none"> <li>• On the local file system</li> <li>• On a local server</li> <li>• On a remote server</li> </ul>
Add Resource Remove Resource Fetch Updates	Located directly under the Resources pane, the function buttons consist of the following: <ul style="list-style-type: none"> <li>• Add Resource – Use to enter the URL to the web service WSDL file.</li> <li>• Remove Resource – Use to remove a URL from the list of web service resources.</li> <li>• Fetch Updates – Use to retrieve XSD and WSDL files for a web service resource. You view these files in the Fetched Resources tab.</li> </ul>
Settings	Use to add a setting for the following: <ul style="list-style-type: none"> <li>• Override URL – Use to enter an override (proxy) URL for a defined web service resource.</li> <li>• Configuration Provider – Use to enter the name of a type that implements the <code>IWsWebServiceConfigurationProvider</code> interface.</li> </ul> See “Adding Configuration Options” on page 72 in the <i>Integration Guide</i> for more information.
Fetched Resources	Shows the XSD and WSDL files pulled from the remote host that are associated with the listed resources.

### See also

- “Calling a BillingCenter Web Service from Java” on page 59 in the *Integration Guide*
- “Loading WSDL Directly into the File System” on page 66 in the *Integration Guide*
- “Adding Configuration Options” on page 72 in the *Integration Guide*
- “Defining a Web Service Collection” on page 116

## Defining a Web Service Collection

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

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

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

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

**To create a web service WSDL**

1. Within Studio, navigate within the **wsi** hierarchy to a package in which to store your files.
2. Right-click and choose **New → Webservice Collection**. Studio prompts you for a name for the web service collection. Enter a name for the web service collection and click **OK**.
3. Click **Add Resource...**
4. Enter the URL of the WSDL for the external web service. This is also called the web service endpoint URL. Studio indicates whether the URL is valid. You cannot proceed until you enter a valid URL. After determining that the URL is valid, click **OK**.
5. Studio indicates that you have modified the list of resource URLs and offers to fetch update resources. Click **Yes**.  
You can click **Fetch Updates** at any time to refresh the WSDL from the web service server.
6. Studio retrieves the WSDL for that service. You see the resource URL in the editor's **Resources** pane.
7. Click **Fetched Resources** to view the WSDL and its associated resource.



# Implementing QuickJump Commands

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

This topic includes:

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

## What Is QuickJump?

The QuickJump box is a text-entry box for entering navigation commands using keyboard shortcuts. Guidewire places the box at the upper-right corner of each BillingCenter 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 BillingCenter. (Therefore, to remove the QuickJump box from the BillingCenter interface, remove all commands from the QuickJump configuration editor.)

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

There are three basic types of navigation commands:

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

## Adding a QuickJump Navigation Command

If you add a command, first set the command type, then define the command by setting certain parameters. The editor contains a table with each row defining a single command and each column representing a specific command parameter. You use certain columns with specific command types only. BillingCenter 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"> <li>• QuickJumpCommandRef</li> <li>• StaticNavigationCommandRef</li> <li>• ContextualNavigationCommandRef</li> </ul>	Display key specifying the command string the user types to invoke the command. Note that the command string must not contain a space.
Command Class	<ul style="list-style-type: none"> <li>• QuickJumpCommandRef</li> </ul>	Class that specifies how to implement the command. This class must be a subclass of QuickJumpCommand. Guidewire intentionally makes the base QuickJumpCommand class package local. To implement, override one of the subclasses described in Implementing QuickJumpCommandRef Commands.
		You only need to subclass QuickJumpCommand if you plan to implement the QuickJumpCommandRef command type. For the other two command types, you use the existing base class appropriate for the command—either StaticNavigationCommand or ContextualNavigationCommand—and enter the other required information in the appropriate columns.
Command Target	<ul style="list-style-type: none"> <li>• StaticNavigationCommandRef</li> <li>• ContextualNavigationCommandRef</li> </ul>	Target page ID.
Command Arguments	<ul style="list-style-type: none"> <li>• StaticNavigationCommandRef</li> </ul>	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"> <li>• ContextualNavigationCommandRef</li> </ul>	Name of a variable on the user's current page.
Context Type	<ul style="list-style-type: none"> <li>• ContextualNavigationCommandRef</li> </ul>	Type of context symbol (variable).

## Implementing QuickJumpCommandRef Commands

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

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

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

### Navigation Commands with One or More Static Parameters

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

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

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

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

### **Navigation Commands with an Explicit Parameter (Including Search)**

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

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

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

### **Navigation Commands with an Inferred Parameter**

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

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

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

### **Navigation to an Entity-Viewing Page**

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

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

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

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

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

## Implementing StaticNavigationCommandRef Commands

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

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

## Implementing ContextualNavigationCommandRef Commands

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

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

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

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

## 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 BillingCenter 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, BillingCenter 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, BillingCenter cannot determine automatically whether to display the command to the user in the `QuickJump` box before the user enters a value for the parameter. If it is possible to manually determine whether to display the command to the user, check for permission using the overridden `isPermitted` method. (This might be, for example, from the destination's `canVisit` attribute.)



# Using the Entity Names Editor

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

This topic includes:

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

## Entity Names Editor

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

BillingCenter 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. BillingCenter uses these entity name definitions in generating database queries that return the limited information needed to construct the display name string. This ensures that BillingCenter does not load the entire entity and its subentities into memory simply to retrieve the few field values necessary to generate the display name.

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

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

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

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

## Variable Table

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

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

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

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

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

---

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

---

### The Entity Path Column

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

Thus:

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

## The Use Entity Names? Column

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

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

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

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

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

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

## Evaluating Null Values

If the value of **Use Entity Name** is **true**, then BillingCenter 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.

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

Therefore, if BillingCenter is in the process of determining how to order two contacts, it first compares the values in the (**Sort Path**) **LastNamesDenorm** fields (**Sort Order** = 1). If these values are equal, Studio then compares the values in the **FirstNamesDenorm** fields (**Sort Order** = 2), repeating this process for as long as there are fields to compare.

These columns specify the default sort order. Other aspects of Guidewire BillingCenter can override this sort order, for example, the sort order property of a list view cell widget.

## Gosu Text Editor

You enter the actual Gosu code used to construct the entity name in the code definition pane underneath the variable table. Studio then replaces the variable with mapped property.

The following Gosu definition code for the Contact entity name shows these mappings.

```
var retString = ""

if ( SubType != null && Person.isAssignableFrom( Type.forName("entity." + SubType) ) ) {

    if (FirstName != null and FirstName.length() > 0) {
        retString = retString + FirstName + " "
    }
    if (LastName != null and LastName.length() > 0) {
        retString = retString + LastName + " "
    }
    if (Suffix != null) {
        retString = retString + gw.api.util.TypeKeyUtil.toDisplayName(Suffix) + " "
    }

} else {
    retString = Name != null and Name.length() > 0 ? Name : ""
}
return retString
```

To use the Contact entity name definition, you can embed the following in a PCF page, for example.

```
<Cell id="Name" value="contact.DisplayName" ... />
```

## Including Data from Subentities

Many times, you want to include information from subentities of the current entity in its Entity Name. For example, this happens often with Contacts related to the current entity. Guidewire recommends that you do one of the following to include data from a subentity. (The two options are mutually exclusive. You must do one or the other.)

### Option 1: Use the DisplayName for a Subentity

To use the `DisplayName` value for a subentity, you must set the value of `Use Entity Name` to `true` on the variable definition. For example, for Contacts, you must set the value to `true` through an explicit `Denorm` column, such as `Exposure.ClaimantDenorm`.

To illustrate:

Name	Entity Path	Use Entity Name?
claimantDisplayName	Exposure.ClaimantDenorm	true
incidentDisplayName	Exposure.Incident	true

### Option 2: Reference Fields on the Subentity

It is possible that you do not want to use the Entity Name as defined for the subentity's type. If so, then you need to set up variables in the table to obtain the fields from the subentity that you need. To illustrate:

Name	Entity Path	Use Entity Name?
claimantFirstName	Exposure.ClaimantDenorm.FirstName	false
claimantLastName	Exposure.ClaimantDenorm.LastName	false
severity	Exposure.Incident.Severity	false
incidentDesc	Exposure.Incident.Description	false

You can then use these variables in Gosu code (in the text editor) to include the Claimant and Incident information in the entity name for Exposure.

#### Guidewire Recommendations

Do not end an Entity Path value with an entity foreign key, without setting the Use Entity Name value to true. Otherwise, BillingCenter 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 BillingCenter 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`. BillingCenter uses the `ContactRoleMessage` type to define the format of the entity name to use in role validation error messages. In some cases, this definition is merely the same as the default definition.

**Note:** It is not possible for you to either add or delete an entity name type from the base application configuration. You can, however, modify the definition—the Gosu code—for all defined types. You can directly access only the default type from Gosu code.



# Using the Messaging Editor

This topic covers how you use the **Messaging** editor in Guidewire Studio.

This topic includes:

- “**Messaging Editor**” on page 131

## Messaging Editor

You use the **Messaging** editor to set up and define one or more message environments, each of which includes one or more message destinations. A message destination is an abstraction that represents an external system. Typically, a destination represents a distinct remote system. However, you can also use destinations to represent different remote APIs or different types of messages that must be sent from BillingCenter 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 315 in the *Integration Guide*
- “Implementing Messaging Plugins” on page 347 in the *Integration Guide*
- “Messaging and Events” on page 303 in the *Integration Guide*

## Adding a Messaging Environment

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

- A development environment
- A test environment

- A production environment

Guidewire provides a single default messaging environment in the BillingCenter base configuration. You see it listed as **Default** in the **Messaging Config** drop-down list.

#### To create a new messaging environment

1. Next to the **Messaging Config** drop-down list, click **Add Messaging** .
2. In the **New Messaging** dialog box, type the name for the new message environment.
3. Add message destinations as required. See “Adding a Message Destination” on page 132 for details.

#### To remove a messaging environment

1. Next to the **Messaging Config** drop-down list, click the messaging environment to remove.
2. Click **Remove Messaging** .

BillingCenter disables the messaging environment **Remove** button if only one message environment exists. However, if there are several messaging environments, and you have added message destinations to each environment, then there are multiple **Remove** options available. The **Remove Messaging** button at the top of the screen removes the currently selected message environment. The other **Remove Destination** option removes the currently selected message destination from the list of destinations.

**Note:** Be careful not to inadvertently click the top **Remove Messaging** button, as BillingCenter deletes the message environment without any additional warning. You cannot undo this action.

## Adding a Message Destination

To add a message destination, open the **Messaging** editor, select a message environment, click **Add Destination**, and fill in the required fields. Notice that Studio requires that you enter a plugin name, for example, for the Transport plugin.

It is important to understand the difference between the implementation class name and the plugin name. After you write code that implements a messaging plugin, you must register it in Studio. As you register an implementation of the new messaging plugin, Studio prompts you for a plugin name. The plugin name is different from the implementation class name. The plugin name is a short arbitrary name that identifies a plugin implementation. Studio only prompts you for a plugin name for plugin interfaces that support more than one implementation. (For example, it is possible to create multiple distinct messaging and encryption plugins.)

After you click **Add Destination** in the **Messaging** editor, fill in the following fields.

<b>ID</b>	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"><li>“Implementing a Message Transport Plugin” on page 348 in the <i>Integration Guide</i></li></ul>
<b>Name</b>	The name to use for this messaging destination.
<b>Transport Plugin</b>	The name of the <code>MessageTransport</code> plugin implementation that knows how to send messages for this messaging destination. A destination must define a message transport plugin that sends a <code>Message</code> object over a physical or abstract transport. For example, the plugin might do one of the following: <ul style="list-style-type: none"><li>Submit the message to a message queue</li><li>Call a remote web service API and get an immediate response that the system handled the message</li><li>Implement a proprietary protocol that is specific to a remote system</li></ul> For more information, see the following: <ul style="list-style-type: none"><li>“Messaging Overview” on page 304 in the <i>Integration Guide</i></li><li>“Implementing a Message Transport Plugin” on page 348 in the <i>Integration Guide</i></li></ul>

If you select a specific row in the message ID table, you see additional fields. These fields have the following meanings:

Field	Description
Request Plugin	<p>A destination can optionally define a message request (<code>MessageRequest</code>) plugin to prepare or pre-process a <code>Message</code> object before a message is sent to the message transport. For example, the <code>MessageRequest</code> plugin can:</p> <ul style="list-style-type: none"> <li>Translate strings or codes in a text-type message payload to codes for a remote system.</li> <li>Translate name/value pairs in a text-type message payload into XML.</li> <li>Set messaging-specific data model extension properties on the <code>Message</code> object before sending it.</li> </ul> <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"> <li>"Implementing a Message Request Plugin" on page 348 in the <i>Integration Guide</i></li> <li>"Message Destination Overview" on page 315 in the <i>Integration Guide</i></li> </ul>
Reply Plugin	<p>A destination can optionally define a message reply (<code>MessageReply</code>) plugin to asynchronously acknowledge a <code>Message</code> object. For instance, this plugin can implement a trigger from an external system to notify BillingCenter 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"> <li>"Implementing a Message Reply Plugin" on page 350 in the <i>Integration Guide</i></li> <li>"Message Destination Overview" on page 315 in the <i>Integration Guide</i></li> </ul>
Chunk Size	<p>The number of messages that the messaging subsystem retrieves from the database in each round of sending, if possible. By default, Guidewire sets this value to 100,000. This number is usually sufficient to include all sendable messages currently in the send queue. Be careful not to set this number too low.</p> <p>For more information, see the following:</p> <ul style="list-style-type: none"> <li>"Message Ordering and Multi-Threaded Sending" on page 337 in the <i>Integration Guide</i></li> </ul>
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"> <li>"Message Ordering and Multi-Threaded Sending" on page 337 in the <i>Integration Guide</i></li> </ul> <p><b>NOTE</b> 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.</p>
Max Retries	The number of retries to attempt before the retryable error becomes non-retryable.
Initial Retry Interval	The amount of time (in milliseconds) to wait before attempting to retry sending a message after a retryable error condition occurs.
Number Sender Threads	BillingCenter does not need to distinguish between safe-ordered and non-safe-ordered messages in the same way that PolicyCenter and ClaimCenter must. Thus, BillingCenter does not support the setting of multiple threads for messaging destinations. This feature is unique to how ClaimCenter and PolicyCenter handle claim-specific (for ClaimCenter) and account-specific (for PolicyCenter) messages. As a result, BillingCenter does not use the <code>Number Sender Threads</code> field in the Studio messaging configuration as ClaimCenter and PolicyCenter do.
	<p>For more information, see the following:</p> <ul style="list-style-type: none"> <li>"Message Ordering and Multi-Threaded Sending" on page 337 in the <i>Integration Guide</i></li> </ul>

Field	Description
Shutdown Timeout	<p>Messaging plugins have methods to handle the administrative commands suspend, resume, and preparing for the messaging system to shutdown. During the suspend, shutdown, and resume methods of the plugin, the plugin must not call any APIs that suspend or resume messaging destinations. (This includes—but is not limited to—IMessageToolsAPI web service APIs.) Doing so creates circular application logic. Guidewire disallows such actions.</p> <p>The shutdown timeout value is the length of time to wait before attempting to shutdown the messaging subsystem.</p> <p>For more information, see the following:</p> <ul style="list-style-type: none"> <li>“Handling Messaging Destination Suspend, Resume, Shutdown” on page 353 in the <i>Integration Guide</i>.</li> </ul>
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, BillingCenter attempts the next retry in 10 minutes.

The **Messaging** editor contains several additional checkboxes directly underneath the message definition fields:

Checkbox	Description
Enabled	Select this checkbox to enable this message destination.
Strict mode	<p>Select this checkbox to ensure that:</p> <ul style="list-style-type: none"> <li>BillingCenter waits for acknowledgements for each non-safe-ordered message before sending the next one.</li> <li>The message sending system blocks all future messages of all types (both safe-ordered and non-safe-ordered) if there are errors in non-safe-ordered messages.</li> </ul> <p>See “Message Ordering and Multi-Threaded Sending” on page 337 in the <i>Integration Guide</i> for more details.</p>

## Associating Event Names with a Message Destination

To define one or more specific events for which you want this message destination to listen, click **Add Event**  under **Events**. Each event triggers the Event Fired rule set for that destination. Use the special event name wild-card string “`(\w)*`” to listen for all events.

To get notifications using Event Fired rules when specific types of data changes occur, you must specify one or more messaging destinations to listen for that event. If no messaging destination listens for an event, BillingCenter 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 303 in the *Integration Guide*.

### See also

- For a list of built-in events that BillingCenter triggers, see “List of Messaging Events in BillingCenter” on page 323 in the *Integration Guide*.

# Using the Display Keys Editor

This topic discusses how to work with the display key editor that is available to you in Guidewire Studio.

This topic includes:

- “Display Keys Editor” on page 135
- “Creating Display Keys in a Gosu Editor” on page 136
- “Retrieving the Value of a Display Key” on page 136

## Display Keys Editor

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

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

`BillingCenter/modules/configuration/config/locale/en_US`

However, if you do localize one or more display keys, then BillingCenter uses additional `display.properties` files, one for each locale that you create. For more information, see “Localizing Display Keys” on page 47 in the *Globalization Guide*.

BillingCenter represents display keys within a hierarchical name space. Within `display.properties`, this translates into a dot (.) separating the levels of the hierarchy.

Within the `Display Keys` editor, Studio does the following automatically:

- It sorts display keys alphabetically—both at the root level and at the package level—as you create the display key.
- It removes empty display keys—those for which no value was set—upon a save operation.

To access the `Display Keys` editor in Studio, in the Project window, navigate to `configuration` → `config` → `Localizations` → `en_US`, and then open the file `display.properties`.

You can also place your cursor in a text string and press Alt+Enter to open the **Create Display Key** dialog.

Using the **Display Keys** editor, you can do the following:

Task	Actions
View a display key	Navigate to the display key that you want to view by scrolling through the <code>display.properties</code> file. To search for a particular key or value, press Ctrl+F and then type your search term in the search bar.
Modify the text of an existing display key	Navigate to the display key that you want to modify, and then modify the string in the editor as you want.
Create a new display key	In the <b>Display Key</b> editor, type the desired name and value for your new display key.
Delete an existing display key	Highlight the display key that you want to delete, and then press Delete.
Localize an existing display key	Select a different locale and enter the localized text. See “ <a href="#">Localizing Display Keys</a> ” on page 47 in the <i>Globalization Guide</i> .

## Creating Display Keys in a Gosu Editor

You can also immediately create a display key from within a Gosu editor by entering a string literal. If you place the cursor within the string and then press Alt+Enter, Studio prompts you to create a new display key for that string.

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

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

If you place the cursor within that string, press Alt+Enter, and then click **Convert string literal to display key**, Studio opens the **Create Display Key** dialog. It also populates the **Display Key Name** field with the string text that you entered. The dialog contains a text entry field in which you can enter the localized text for the string that you entered in the Rules editor.

After you enter the text and click **OK**, Studio replaces the string literal with the new display key. For example:

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

## Retrieving the Value of a Display Key

Some display keys contain a place holder argument or parameter, designated by `{0}`. BillingCenter 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, BillingCenter replaces `{0}` with the appropriate value, in this case, the name of an activity.

Occasionally, there are display keys that contain multiple arguments. For example:

```
Java.Admin.User.InvalidGroupAdd = The group {0} cannot be added for the user {1}
as they do not belong to the same organization.
```

### Class `displaykey`

Use the `displaykey` class to return the value of the display key. Use the following syntax:

```
displaykey.[path to display key].[display key name]
```

For example:

```
displaykey.Java.Admin.User.DuplicateRoleError
```

returns

```
User has duplicate roles
```

This also works with display keys that require a parameter or parameters. To retrieve the parameter value, use the following syntax.

```
displaykey.[path to display key].[display key name](arg1)
```

For example, file `display.properties` defines the following display key with placeholder `{0}`:

```
Java.UserDetail.Delete.IsSupervisorError = Cannot delete user because that user is the supervisor  
of the following groups\: {0}
```

Suppose that you have the following display key code:

```
displaykey.Java.UserDetail.Delete.IsSupervisorError( GroupName )
```

If you have already retrieved a value for `GroupName`, this display key returns the following:

```
Cannot delete user because they are supervisor of the following groups: WesternRegion
```

The same syntax works with multiple arguments as well:

```
displaykey.[path to display key].[display key name](arg1, arg2, ...)
```



# Data Model Configuration



# Working with the Data Dictionary

Guidewire provides the *Data Dictionary* to help you understand the BillingCenter 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.

This topic includes:

- “What is the Data Dictionary?” on page 141
- “What Can You View in the Data Dictionary?” on page 142
- “Using the Data Dictionary” on page 142

## What is the Data Dictionary?

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

```
gwbc regen-dictionary
```

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

```
BillingCenter/build/dictionary/data/
```

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

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

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

**See also**

- “Regenerating the Data Dictionary and Security Dictionary” on page 24

## What Can You View in the Data Dictionary?

**Note:** If you use a third-party tool to edit BillingCenter 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 `BillingCenter/build/dictionary/data/index.html`), Guidewire presents you with multiple choices. For example, you can choose to view either **Data Entities** or **Data Entities (Migration View)**.

The standard and migration views are similar but not identical. You use each for a different purpose. In general:

- Use the *standard view* to view a full set of entities associated with the BillingCenter application and the columns, typekeys, arrays and foreign keys associated with each entity. “Using the Data Dictionary” on page 142 discusses the standard *Data Dictionary* view in more detail.
- Use the *migration view* to assist you in converting data from a legacy application. This view provides a subset of the information in the standard view of the application entities that is more useful for those working on the conversion of legacy data.

### The Migration View of the Data Dictionary

The standard *Data Dictionary* view separates out entity subtypes from the main entity supertype. In brief, a *supertype* relates to a *subtype* in a parent-child relationship. For example, if a **Contact** data entity is the supertype, then **Person** and **Company** are examples of its subtypes. Thus, an entity subtype inherits the characteristics of its supertype and adds individual variations particular to it.

This separation into supertype and subtype is not particularly useful for data conversion (the process of importing data into BillingCenter from an external legacy application). Therefore, the migration view of the *Data Dictionary* differs from the standard view in the following respects:

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

## Using the Data Dictionary

You use the *Data Dictionary* to do the following:

- To determine what a field means that you see in a data view definition.

- To see what fields are available to add to a view, or to use in rules, or to export in an integration template, and more.
- To view the list of options for an associated typekey field. (See “What is a Typelist?” on page 246 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**, BillingCenter displays a left-side pane listing all of the entities in BillingCenter. Then, on the right-side, BillingCenter displays a pane that shows the details of the selected item in the left-side pane.

Within the details of an object, you can follow links to related objects or view the allowed values for a typelist.

The following topics describe:

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

## Field Colors

An examination of the *Data Dictionary* shows fields in green, blue, and red. These colors have the following meanings:

Color	Meaning
Green	<p>The object field (column) is part of the Guidewire base configuration. The object definition file exists in Studio in the following locations:</p> <ul style="list-style-type: none"><li>• config → configuration → Metadata</li><li>• config → configuration → Extensions</li></ul>
Blue	<p>The object field (column) is defined in an extension file, either by Guidewire or as a user customization. The object definition file exists in Studio in the following location:</p> <ul style="list-style-type: none"><li>• config → configuration → Extensions</li></ul> <p>It is possible for Guidewire to define a base object in the <b>Metadata</b> folder, and then to extend the object using an extension entity in the <b>extensions</b> folder.</p>
Red	<p>Occasionally, it is possible to see a message in red in the Data Dictionary that states:</p> <p><i>This entity is overwritten by the application during staging.</i></p> <p>This message indicates that Guidewire BillingCenter auto-populates a table or column's staging table equivalent. Do not attempt to populate the table yourself as the loader import process overwrites the staging table during import.</p> <p>See also the description of the <code>overwrittenInStagingTable</code> attribute in “Entity Data Objects” on page 161.</p>

## Object Attributes

An object in the BillingCenter 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 **Account** entity has the attributes **Editable**, **Extendable**, **Final**, **Keyed**, **Loadable**, **Sourceable**, and **Versionable**.

The following list describes the possible attributes:

Attribute	Description
Abstract	The entity is a supertype. However, all instances of it must be one of its subtypes. That is, you cannot instantiate the supertype entity itself. An abstract entity is appropriate if the supertype serves only to collect logic or common fields, but does not make sense to exist on its own.
Editable	The related database table contains rows that you can edit. An Editable table manages additional fields that track the immediate status of an entity in the table. For example, it tracks who created it and the time, and who last edited it and the time.
Extendable	It is possible to extend the entity with additional custom fields added to it.
Final	It is not possible to subtype this entity. You can, however, extend it by adding fields to it.
Keyed	The entity has a related database table that has a primary key. Each row in a Keyed table has an integer primary key named ID. BillingCenter 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. BillingCenter deletes the entity after the operation is complete.
Versionable	The entity has a version number that increases every time the entity changes. The BillingCenter cache uses the version number to determine if updates have been made to an entity.

To view the definition of a particular attribute, click the tiny question mark (?) by the attribute name in the attribute list in the *Guidewire Data Dictionary*.

## Entity Subtypes

If you look at Contact in the *Guidewire Data Dictionary*, for example, you see that data dictionary lists a number of subtypes. For certain BillingCenter 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.

BillingCenter 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.

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

## Virtual Properties on Data Entities

The *Data Dictionary* lists certain entity properties as *virtual*. BillingCenter does not store virtual properties in the BillingCenter physical database. Instead, it derives a virtual property through a method, a concatenation of other fields, or from a pointer (foreign key) to a field that resides elsewhere.

For example, if you view the Account entity in the *Data Dictionary* (for PolicyCenter), you see the following next to the **AccountContactRoleSubtypes** field:

```
Derived property returning gw.api.database.IQueryResult (virtual property)
```

### Examples

The following examples illustrate some of the various ways that Guidewire applications determine a virtual property. The following examples use Guidewire ClaimCenter for illustration.

#### Virtual Property Based on a ForeignKey

`Claim.BenefitsDecisionReason` is a virtual property that simply pulls its value from the `cc_claimtext` table, which stores `ClaimText.ClaimTextType = BenefitsDecisionReason`. It returns a `mediumtext` value. The other fields in `cc_claimtext` and `cc_exposuretext` work in a similar fashion.

#### Virtual Property Based on an Associated Role

`Claim.claimant` is a virtual property that retrieves the `Contact` associated with the `Claim` with the `ClaimContactRole` of `claimant`. It returns a `Person` value.

#### Virtual Property Based on a Typelist

`Contact.PrimaryPhoneValue` is a virtual property that calculates its return value based on the value from `Contact.PrimaryPhone`. It retrieves the telephone number stored in the field represented by that typekey. This can be one of the following:

- `Contact.HomePhone`
- `Contact.WorkPhone`
- `Person.CellPhone`

It returns a phone value.



# The BillingCenter Data Model

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

This topic includes:

- “What is the Data Model?” on page 147
- “Overview of Data Entities” on page 149
- “Base BillingCenter Data Objects” on page 158
- “Data Object Subelements” on page 174

## What is the Data Model?

At its simplest, the Guidewire data model is a set of XML-formatted metadata definitions of entities and type-lists.

<b>Entities</b>	An <i>entity</i> defines a set of fields for information. You can add the following kinds of fields to an entity: <ul style="list-style-type: none"><li>• Column</li><li>• Type key</li><li>• Array</li><li>• Foreign key</li><li>• Edge foreign key</li></ul>
<b>TypeLists</b>	A <i>typelist</i> defines a set code/value pairs, called <i>typecodes</i> , that you can specify as the allowable values for the type key fields of entities. Several levels of restriction control what you can modify in typelists: <ul style="list-style-type: none"><li>• <b>Internal typelists</b> – You cannot modify internal typelists because the application depends upon them for internal application logic.</li><li>• <b>Extendable typelists</b> – You can modify this kind of typelist according to its schema definition.</li><li>• <b>Custom typelists</b> – You can also create custom typelists for use on new fields on existing entities or for use with new entities.</li></ul>

Guidewire BillingCenter loads the metadata of the data model on start-up. The loaded metadata instantiates the data model as a collection of tables in the application database. Also, the loaded metadata injects Java and Gosu classes in the application server to provide a programmatic interface to the entities and typelists in the database.

## The Data Model in Guidewire Application Architecture

Guidewire applications employ a metadata approach to data objects. BillingCenter uses metadata about application domain objects to drive both database persistence objects and the Gosu and Java interfaces to these objects.

This architecture provides enormous power to extend Guidewire application capabilities. Typically, you alter enterprise-level software applications through customization, wherein you change the behavior of the software by editing the code itself. In contrast, a Guidewire application uses XML files that provide default behavior, permissions and objects in the base configuration. You change the behavior of the application by modifying the base XML files and by creating Gosu business rules, classes, enhancements, and other objects.

## The Base Data Model

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

For example, the BillingCenter data model defines a `Payment` entity and several fields on it such, as `Account`, `CheckNumber`, and `Status`.

BillingCenter lets you change its data model to accommodate your business needs. You make your changes to the data model by modifying existing XML files and adding new ones. BillingCenter stores your files that change the data model in the following application directory:

`BillingCenter/modules/configuration/config/extensions`

However, you always access and edit the data model files indirectly through the `configuration → config → Extensions` folder in Studio. Do not edit the XML files directly from the file system yourself.

Guidewire calls changes that you make to the data model *data model extensions*. For example, you can extend the data model by adding new fields to the `User` entity, or you can declare entirely new entities. The complete data model of your BillingCenter installation comprises the BillingCenter model and any data model extensions that you make.

---

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

---

## Working with Dot Notation

Many places within BillingCenter require knowledge of fields within the application data model, especially while you configure BillingCenter. For example, code in a business rule, class or enhancement may need to check the *owner* of an assignable object. Or, code may need to check the date and time of object creation.

BillingCenter provides an easy and consistent method of referring to fields within the data model, using relative references based on a *root object*.

A root object is the starting point for any field reference. If you run Gosu rules on a trouble ticket for example, the trouble ticket is the root object and you can access anything that relates to this trouble ticket. 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 `User` associated with the activity.

Guidewire applications use *dot notation* for relative references. For example, assume that your Guidewire application has `TroubleTicket` as the root object. For a simple reference to a field on the trouble ticket such as the update time, you simply use:

```
troubleTicket.UpdateTime
```

However, suppose that you want to reference a field on an entity that relates to the trouble ticket, such as a back-up user for the assigned *owner*. You must first describe the path from the trouble ticket to the assigned user, then describe the path from the assigned user to the back-up user:

```
troubleTicket.AssignedUser.BackupUser
```

## Overview of Data Entities

Data entities are the high-level business objects used by BillingCenter, such as a `TroubleTicket` or `Disbursement`. An entity serves as the root object for data views, rules, Gosu classes, and most other data-related areas of BillingCenter. Guidewire defines a set of data objects in the base BillingCenter configuration from which it derives all other objects and entities. For many of the Guidewire base entities, you can also create entity extensions that enhance the base entities and provide additions required to support your particular business needs. In some cases, you can even define entirely new entities.

### Data Entity Metadata Files

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

**WARNING** Do not modify any of the base data entity definition files (those in the `modules/configuration/config/metadata` directory) by editing them directly. You can view these files in read-only mode in Studio in the `configuration → config → Metadata` folder.

To better understand the syntax of entity metadata, it is sometimes helpful to look at the BillingCenter data model and its metadata definition files. BillingCenter uses separate metadata definition files for entity declarations and extensions to them.

The base metadata files are available in Studio in the following location: `configuration → config → Metadata`

The extension metadata files are available in Studio in the following location: `configuration → config → Extensions`

The file extensions of metadata definition files distinguish their type, purpose, and contents.

File type	Purpose	Contains	More information
.dti	Data Type Info	A single data type definition.	"Data Types" on page 225
<b>Entities</b>			
.eti	Entity Type Information	A single Guidewire or custom entity declaration. The name of the file corresponds to the name of the entity being declared.	"Base BillingCenter Data Objects" on page 158
.eix	Entity Internal eXtension	A single Guidewire entity extension. The name of the file corresponds to the name of the Guidewire entity being extended.	Internal defintion; see .etx
.etx	Entity Type eXtension	A single Guidewire or custom entity extension. The name of the file corresponds to the name of the entity being extended.	"Extension Data Objects" on page 166 "View Entity Extension Data Objects" on page 173
<b>Typelists</b>			
.tti	Typelist Type Info	A single Guidewire or custom typelist declaration. The name of the file corresponds to the name of the typelist being declared.	"Working with TypeLists" on page 245

File type	Purpose	Contains	More information
.tix	Typelist Internal eXtension	A single Guidewire typelist extension. The name of the file corresponds to the name of the Guidewire typelist being extended.	"Working with Typelists" on page 245
.txt	Typelist Type eXtension	A single Guidewire or custom typelist extension. The name of the file corresponds to the name of the typelist being extended.	"Working with Typelists" on page 245

The type of a metadata definition file determines what you can store and whether you can modify its contents.

File type	Location	Files are modifiable
.dti	configuration → config → datatypes	No
<b>Entities</b>		
.eti	configuration → config → Extensions → Entity	Yes
	configuration → config → Metadata → Entity	No
.eix	configuration → config → Metadata → Entity	No
.etx	configuration → config → Extensions → Entity	Yes
<b>Typelists</b>		
.tti	configuration → config → Extensions → Typelist	Yes
	configuration → config → Metadata → Typelist	No
.tix	configuration → config → Metadata → Typelist	No
.txt	configuration → config → Extensions → Typelist	Yes

## The Metadata Folder

The **Metadata** folder contains the metadata definition files for entities that comprise the BillingCenter data model.

A **Metadata** folder contains the following metadata definition file types:

- **Declaration files** – Versions of metadata definition files with extensions \*.eti and \*.tti.
- **Internal extension files** – Versions of metadata definition files with extensions \*.eix or \*.tix.

For an example, the BillingCenter data model includes the following metadata definition files that collectively define the Address entity type.

File version	Metadata location	File purpose
Address.eti	configuration → config → Metadata → Entity	Entity definition
Address.eix	configuration → config → Metadata → Entity	Extension to the entity definition

At runtime, Guidewire merges the .eti and .eix versions of the Address definition file to create a complete BillingCenter Address entity type.

## The Extensions Folder

The **configuration → config → Extensions** folder contains your data model definitions that extend the BillingCenter data model. BillingCenter considers the base definitions in **configuration → config → Metadata** first, and then applies the definitions in the **Extensions** folder to them. This lets you create an entity extension that overrides any Guidewire entity extensions.

## Example of Activity Metadata and Extension Files

The BillingCenter data model includes the following metadata definition files that collectively define the BillingCenter **Activity** entity.

File	Location	Purpose
Activity.eti	configuration → config → Metadata → Entity	Entity definition, not modifiable.
Activity.eix	configuration → config → Metadata → Entity	Entity extension, not modifiable.

To extend the BillingCenter **Activity** entity, create the following extension file through Guidewire Studio.

File	Location	Purpose
Activity.etc	configuration → config → Extensions → Entity	Custom entity extension.

**WARNING** Use only Guidewire Studio to create data model definition files. Use of Studio assures that the files reside in the correct location.

### See also

- For information on how Guidewire BillingCenter creates merged virtual directories and the directory hierarchy in general, see “Setting Font Display Options” on page 93.

## The `extensions.properties` File

In general, if you change the data model by creating custom data model extensions in `configuration → config → Extensions`, BillingCenter automatically upgrades the database the next time that you start the application server. It detects changes to files in that directory by recording a checksum each time the application server starts. If the recorded checksum and the current checksum differ, BillingCenter upgrades the database.

Sometimes you want to force BillingCenter to upgrade the database without making changes to your custom data model extensions. The `configuration → config → Extensions` folder in Studio contains an `extensions.properties` file that contains the numeric property `version`. The value of the `version` property represents the current version of the data model definition for your instance of BillingCenter. It controls whether BillingCenter performs a database upgrade on server startup.

Whenever BillingCenter upgrades the database, BillingCenter stores the value of the `version` property in the database. The next time the application server starts up, BillingCenter compares the value of the property in the database to the value in the `extensions.properties` file. If the value in the database is lower than the value in the file, BillingCenter performs a database upgrade. If the value in the database is higher, the upgrade fails.

**WARNING** In a production environment, Guidewire requires that you increment the version number whenever you make changes to the data model before you restart the application server. Otherwise, unpredictable results can occur. Use of the `extensions.properties` file in a development environment is optional.

## Working with Entity Definitions

In working with entity definitions, you typically want to perform the following operations:

- Search for an existing entity definition
- Create a new entity definition
- Extend an existing entity definition

This section describes procedures for each operation.

**Note:** Guidewire strongly recommends that you verify your entity definitions at the time that you create them. To do so, right-click the entity in the Project window, and then click **Validate**. The verification process highlights any issues with a data model definition, enabling you to correct any issues.

## Search for an Existing Entity Definition

1. In the Project window, press **Ctrl+N**.

The **Enter class name** dialog opens.

2. Type the name of the entity that you want to find.

Studio displays a list of matching entries that start with the character string that you typed.

3. In the list, click the name of the entity definition that you want to view.

Pay attention to the class type. For example, if you type “Activity”, Studio displays a list that includes all components whose name contains that text. Look for the one that says **(entity)** after it.

### Result

Studio opens the file in its editor.

## Create a New Entity Definition

1. In the Project window, navigate to **configuration** → **config** → **Extensions** → **Entity**.

2. Right-click **Entity**, and then click **New** → **Entity**.

3. In the **Entity** text box, type the name of the new entity definition that you want to create. Set the other properties for the entity.

4. Click **OK**.

### Result

Studio displays the name of your new file in the **Extensions** → **entity** folder in Studio, and it stores the new file in the file system at the following location.

`configuration/config/extensions/entity`

Then, Studio opens your new file in its editor.

## Extend an Existing Entity Definition

You can extend only entity definition files that have the **.eti** extension.

### To extend an existing entity definition

1. In the Project window, navigate to **configuration** → **config** → **Metadata**, and then expand **Entity**.

2. Right-click the entity that you want to extend, and then click **New** → **Entity Extension**.

The file that you want to extend must have the **.eti** extension.

3. In the **Entity Extension** dialog, Studio displays the name and location of the extension file it will create. Click **OK**.

### Result

Studio displays the name of your new file in the **Extensions** → **entity** folder and stores the new file at the following location.

`configuration/config/extensions/entity`

Studio then opens your new file in its editor.

## BillingCenter Data Entities

BillingCenter uses XML metadata files to define all data entities in the data model. The `datamodel.xsd` file defines the elements and attributes that you can include in the XML metadata files. You can view a read-only version of this file in the `configuration → xsd → metadata` folder in Studio.

**WARNING** Do not attempt to modify `datamodel.xsd`. You can invalidate your BillingCenter installation and prevent it from starting thereafter.

File `datamodel.xsd` defines the following:

- The set of allowable or valid data entities
- The attributes associated with each data entity
- The allowable subelements on each data entity

All BillingCenter entity definition files must correspond to the definitions in `datamodel.xsd`.

Using XML files, Guidewire defines a data entity as a root element in an XML file that bears the name of the entity. For example, Guidewire declares the `Activity` entity type with the following `Activity.eti` file:

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
        desc="An activity is a instance of work assigned to a user and belonging to a claim."
        entity="Activity"
        exportable="true"
        extendable="true"
        javaClass="com.guidewire.pl.domain.activity.ActivityBase"
        platform="true"
        table="activity"
        type="retireable">
    ...
</entity>
```

At application server start up, BillingCenter loads the XML definitions of the data entities into the application database.

## Data Entities and the Application Database

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

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

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

- If a data entity has a `type` value of `versionable`, BillingCenter stores instances of the entity in the database with a specific ID and version number.

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

---

**IMPORTANT** For each data entity in the BillingCenter data model and for each entity type that you declare, BillingCenter automatically generates a field named `ID` that is of data type `key`. An `ID` field is the internally managed primary key for the object. Do not attempt to create entity fields of type `key`. The `key` type is for Guidewire internal use only. Guidewire also reserves the exclusive use of the following additional data types: `foreignkey`, `typekey`, and `typeListkey`.

---

The following table lists the possible values for the entity type attribute. Use only those type attributes marked for general use to create or extend an data entity. Do not attempt to create or extend an entity with a type attribute marked for internal-use.

Type attribute	Usage	Description
<code>editable</code>	Internal use	An <code>editable</code> entity is a <code>versionable</code> entity. BillingCenter 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"><li>• <code>CreateUser</code> and <code>CreateTime</code></li><li>• <code>UpdateUser</code> and <code>UpdateTime</code></li></ul>
<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 type attribute of <code>nonkeyable</code>.</i>

Type attribute	Usage	Description
retireable	General use	<p>The <code>retireable</code> entity is an extension of the <code>editable</code> entity, and is the most common type of entity. Most, but not all, base entities are of this type.</p> <p>After BillingCenter adds an instance of a <code>retireable</code> data entity to the database, BillingCenter never deletes the instance. Instead, BillingCenter retires the instance. For example, if you select a <code>retireable</code> instance in a list view and then click <b>Delete</b>, BillingCenter preserves the instance in the database. However, BillingCenter inserts an integer in the <code>Retired</code> column for the row that represents the instance. Any non-zero value in the <code>Retired</code> column indicates that BillingCenter considers the instance retired.</p> <p>BillingCenter automatically creates the following fields for <code>retireable</code> entities:</p> <ul style="list-style-type: none"> <li>• <code>ID</code> and <code>PublicID</code></li> <li>• <code>CreateUser</code> and <code>CreateTime</code></li> <li>• <code>UpdateUser</code> and <code>UpdateTime</code></li> <li>• <code>Retired</code></li> <li>• <code>BeanVersion</code></li> </ul> <p>These are the same fields as those BillingCenter creates for <code>editab1e</code> entities, with the addition of <code>Retired</code> property.</p> <p><b>IMPORTANT</b> Although it is extremely common for a base entity to be retireable, it is not required. You cannot assume this to be the case. Always check the <i>Data Dictionary</i> to determine the retireability of an entity.</p>
versionable	General use	<p>An entity that has a version and ID. Entities of this type can detect concurrent updates. In general practice, Guidewire recommends that you use this entity type instead of <code>keyable</code>. <code>Versionable</code> extends <code>keyable</code>.</p> <p>It is possible to delete entities of this type from the database.</p>

## BillingCenter Database Tables

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

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

<code>bc_</code>	Entity table – one for each entity in the base configuration
<code>bcx_</code>	Extension table – one for each custom extension added to the <code>extensions</code> folder in Studio

**Note:** It is possible to create non-persistent entities. These are entities or objects that you cannot save to the database. Guidewire discourages the use of non-persistent entities in favor of Plain Old Gosu Objects (POGOs), instead. See “Non-persistent Entity Data Objects” on page 167 for more information.

Besides entity tables, BillingCenter creates the following types of tables in the database:

- Shadow tables
- Staging tables
- Temporary (temp) tables

### Shadow Tables

A shadow table stores a copy of data from a main table for testing purposes. Every entity table potentially has a corresponding shadow table. Shadow tables in the database have one of the following prefixes:

<code>bct_</code>	Entity shadow table
<code>bctt_</code>	Typelist shadow table

BillingCenter creates shadow tables at server startup only if before you start the server you set the `server.running.tests` system property to `true` explicitly or programmatically.

Shadow tables provide a way to quickly save and restore test data. All GUnit tests, including those that you write yourself, use shadow tables automatically. You cannot prevent GUnit tests from using shadow tables. GUnit tests use shadow tables according to the following process

1. GUnit copies data from the main application tables to the shadow tables to create a backup your test data.
2. GUnit runs your tests.
3. GUnit copies data backed up data in shadow tables to the main tables to restore a fresh copy of your test data for subsequent tests.

## Staging Tables

BillingCenter generates a staging table for any entity that is marked with an attribute of `loadable="true"`. The `loadable` attribute is `true` by default in the base configuration. A staging table largely parallels the main entity table except that:

- BillingCenter replaces foreign keys by `PublicID` objects of type `String`.
- BillingCenter replaces typecode fields by `typekey` objects of type `String`.

After you load data into these staging tables, you run the command line tool `table_import` to bulk load the staging table data into the main application database tables. See “Table Import Command” on page 195 in the *System Administration Guide* for information on use this command.

---

**IMPORTANT** Some data types, for example, `Entity`, contain an `overwrittenInStagingTable` attribute. If this attribute is set to `true`, then do not attempt to populate the associated staging table yourself because the loader import process overwrites this table.

---

In the application database, you can identify a staging table by the following prefix `bcst_`.

## Temporary (Temp) Tables

BillingCenter generates a temporary table for any entity that is marked with an attribute of `temporary="true"`. Do not confuse a temporary table with a shadow table, they are not synonymous. BillingCenter uses temporary tables as work tables during installation or upgrade only. BillingCenter does not use them if the server is running in standard operation.

Unfortunately, it is easy to forget to clear up these tables if they are no longer needed. Therefore, it is quite possible for an application to have several of these temporary tables remaining even though the upgrade triggers that used them are long gone.

In the application database, temporary tables look like any other entity table except that temporary tables are almost always empty.

## Data Objects and Scriptability

Guidewire defines *scriptability* as the ability of code to *set* (write) or *get* (read) a scriptable item such as a property (column) on an entity. To do so, you set the following attributes:

- `getterScriptability`
- `setterScriptability`

The following table lists the different types of scriptability:

Type	Description
all	Exposed in Gosu, wherever Gosu is valid, for example, in rules and PCF files
doesNotExist	Not exposed in Gosu
hidden	Not exposed in Gosu

If you do not specify a scriptability annotation, then BillingCenter defaults to a scriptability of all.

**IMPORTANT** There are subtle differences in how BillingCenter treats entities and fields marked as doesNotExist and hidden. However, these differences relate to internal BillingCenter code. For your purpose, these two annotations behave in an identical manner, meaning any entity or field that uses one of these annotations does not show in Gosu code. In general, there is no need for you to use either one of these annotations.

### Scriptability Behavior on Entities

If you set `setterScriptability` at the entity level but you also set the value to `hidden` or `doesNotExist`, then Guidewire does not generate constructors for the entity. In essence, you cannot create a new instance of the entity in Gosu. Within the BillingCenter data model, you can set the following scriptability annotation on `<entity>` objects:

Object	Set (write)	Get (read)
<code>&lt;entity&gt;</code>	Yes	No <sup>1</sup>
1. <code>&lt;entity&gt;</code> does not contain a <code>getterScriptability</code> attribute.		

### Scriptability Behavior on Fields (Columns)

If you set `setterScriptability` at the field level, then the value that you set controls the writability of the associated property in Gosu. Within the BillingCenter data model, you can set the following scriptability annotation on fields on `<entity>` objects:

Field	Set (write)	Get (read)
<code>&lt;array&gt;</code>	Yes	Yes
<code>&lt;column&gt;</code>	Yes	Yes
<code>&lt;edgeForeignKey&gt;</code>	Yes	Yes
<code>&lt;foreignkey&gt;</code>	Yes	Yes
<code>&lt;onetooone&gt;</code>	Yes	Yes
<code>&lt;typekey&gt;</code>	Yes	Yes

## Base BillingCenter Data Objects

All BillingCenter 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 BillingCenter configuration.

Data object	Extension	Folder	More information
<delegate>	.eti	metadata, extensions	"Delegate Data Objects" on page 158
<entity>	.eti	metadata, extensions	"Entity Data Objects" on page 161
<extension>	.etx	extensions	"Extension Data Objects" on page 166
<nonPersistentEntity>	.eti	metadata, extensions	"Non-persistent Entity Data Objects" on page 167
<subtype>	.eti	metadata, extensions	"Subtype Data Objects" on page 169
<viewEntity>	.eti	metadata	"View Entity Data Objects" on page 171
<viewEntityExtension>	.etx	extensions	"View Entity Extension Data Objects" on page 173

**IMPORTANT** There are additional data objects that Guidewire uses for internal purposes. Do not attempt to create or extend a data entity that is not listed in the previous table.

## Delegate Data Objects

A delegate data object is a reusable entity that contains an interface and a default implementation of that interface. A delegate may also add its own columns to the tables of data objects that implement the delegate. This type of delegation enables a data object to implement an interface while delegating the implementation to the delegate.

You often use a delegate so objects can share code. The delegate implements the shared code rather than each class implementing copies of common code. Thus, a delegate is an entity associated with an implemented interface that multiple parent entities can reuse.

Guidewire defines delegate data object in data model metadata files as the <delegate> XML root element. You can extend existing delegates that are marked as extendable, and you can create your own delegates.

BillingCenter stores all database columns on the delegate entity on the parent entity.

### Implementing Delegate Objects

To implement most delegate objects, you add the following to an entity definition or extension.

```
<implementsEntity name="SomeDelegate"/>
```

For example, in the base configuration, the Group entity implements the Validatable delegate by using the following:

```
<entity entity="Group" ... >
  <implementsEntity name="Validatable"/>
  ...
</entity>
```

It is possible for an entity to implement multiple delegates, just as a Gosu or Java class can implement multiple interfaces.

**See also**

- “<implementsEntity>” on page 188

## Delegate Objects That You Cannot Implement Directly

There are some delegates that you cannot implement directly through the use of the <implementsEntity> element. They are:

- Versionable
- KeyableBean
- Editable
- Retireable

These are special delegates that BillingCenter implicitly adds to an entity if you set the type attribute on the entity to one of these values. Therefore, do not use the <implementsEntity> element to specify one of these delegates. Instead, use the type attribute on the entity declaration. The basic syntax looks similar to the following:

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

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

```
<entity entity="Group" ... type="retirable">
  <implementsEntity name="Validatable"/>
  ...
</entity>
```

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

**See also**

- For an example of how to create a delegate object, see “Creating a New Delegate Object” on page 212.
- For a discussion of working with delegates in Gosu classes, see “Using Gosu Composition” on page 223 in the *Gosu Reference Guide*.

### Attributes of <delegate>

The <delegate> element contains the following attributes.

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

<delegate> attribute	Description	Default
extendable	<i>Internal.</i>	false
name	<i>Required.</i>	None
platform	<i>Internal.</i> Do not use. The only real effect is to change the location in which the table appears in a data distribution report.	false

<delegate> attribute	Description	Default
requires	<p><i>Optional.</i> Specifies an interface for which the implementers of this delegate must provide an implementation. An <i>implementer</i> is an entity that specifies the delegate by using &lt;implementsEntity&gt;.</p> <p><b>IMPORTANT</b> This attribute is inter-related with the adapter attributes of &lt;implementsEntity&gt;.</p> <ul style="list-style-type: none"> <li>If you specify a value for the requires attribute, then the implementers of this delegate must specify a value for the adapter attribute on &lt;implementsEntity&gt;. 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.</li> <li>If you do not specify a value for the requires attribute, then the implementers must not specify an adapter attribute on &lt;implementsEntity&gt;.</li> </ul>	None
subpackage	Sub-package to which the class corresponding to this entity belongs.	None

### Subelements of <delegate>

The <delegate> element contains the following subelements.

<delegate> subelement	Description
column	See “<column>” on page 178.
datetimeordering	<i>Internal.</i>
dbcheckbuilder	<i>Internal.</i>
foreignkey	See “<foreignkey>” on page 186.
fulldescription	See “<fulldescription>” on page 188.
implementsEntity	See “<implementsEntity>” on page 188.
implementsInterface	See “<implementsInterface>” on page 189.
index	See “<index>” on page 190.
monetaryamount	Handles monetary amounts. The <monetaryamount> subelement is a compound data type that stores its values in two separate database columns: a <money> column type, and a typekey to the Currency typelist.
typekey	See “<typekey>” on page 194.

## 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
- Editable
- Validatable
- ...

To determine the list of base configuration delegate entities, search the **configuration → config → Metadata → Entity** folder for files that contain the following text:

```
<delegate
```

### 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. BillingCenter stores information on all subtypes of a supertype entity in a single table. This can create a table that is extremely large and extremely wide. This is true especially if you have an entity hierarchy with a number of different subtypes that each have their own columns. Using a delegate avoids this single-table inheritance while preserving the ability to define the fields and behavior common to all the subtypes in one place.

Guidewire recommends that you consider carefully before making a decision on how to model your entity hierarchy.

### Using Entity Polymorphism

Guidewire recommends that you create a delegate entity if you want to use polymorphism on class methods. For core BillingCenter classes defined in Java, you cannot override these class methods on its Gosu subtypes. You can, however, push all methods and behaviors that can possibly be polymorphic into an interface, rather than the Java superclass. You can then require that all implementers of the delegate implement that interface (the `<implementsEntity>`) through the use of the delegate `requires` attribute. This delegate usage permits the use of polymorphism and enables delegate implementations to share common implementations on a common superclass.

## Entity Data Objects

An entity data object is the standard persistent data object that defines many—if not most—of the BillingCenter entities. Guidewire defines this object in the data model metadata files as the `<entity>` XML root element.

### Attributes of `<entity>`

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

<code>&lt;entity&gt;</code> attribute	Description	Default
<code>abstract</code>	If true, you cannot create an instance of the entity type at runtime. Instead, you must declare a subtype entity with <code>abstract=false</code> , which you can instantiate. Any of the generated code is abstract.	false
<code>admin</code>	Determines whether you can reference the entity from staging tables: <ul style="list-style-type: none"> <li>Entity X has <code>admin="true"</code>. Suppose that you have another, loadable table Y that has a foreign key to X. Then at the time you load the staging table for Y, you can load public IDs that specify entities of type X that are already in the main tables.</li> <li>Entity X has <code>admin="false"</code>. Any Y that you load into a staging table must specify an X that is being loaded into the staging table for X at the same time.</li> </ul> This is important because it allows the staging table loader to do less checking at load time. For example: <ul style="list-style-type: none"> <li>If <code>admin="false"</code>, then the staging table loader merely has to check that all public IDs in <code>ccst_y</code> specify valid entries in <code>ccst_x</code>.</li> <li>If <code>admin="true"</code>, then the staging table loader has to check that all public IDs in <code>ccst_y</code> specify a valid entry in <code>ccst_x</code>. It must also check that all public IDs in <code>ccst_y</code> specify a valid entry in <code>cc_x</code>, the main table.</li> </ul>	false
<code>cacheable</code>	<i>Internal.</i> If set to <code>false</code> , then Guidewire prohibits entities of this type and all its subtypes from existing in the global cache.	true

<entity> attribute	Description	Default
consistentChildren	<p><i>Internal.</i> If set to true, then BillingCenter generates a consistency check and a loader validation that tries to ensure that links between child entities of this entity are consistent. Guidewire enforces the constraint only while loading data from staging tables. You can detect violations of the constraint on data committed to entity tables after the fact by running a consistency check.</p> <p><b>IMPORTANT</b> Guidewire does not enforce consistentChildren constraints at bundle commit.</p>	false
desc	A description of the purpose and use of the entity.	None
displayName	<p><i>Optional.</i> Creates a more human-readable form of the entity name. You can access this name using the following:</p> <pre data-bbox="646 572 850 599">entity.DisplayName</pre> <p>If you do not specify a value for the DisplayName attribute, then the <code>entity.DisplayName</code> method returns the value of the <code>entity</code> attribute, instead. If you subtype an entity that has a specified display name, then the <code>entity.DisplayName</code> method returns the name of the subtype key.</p>	None
edgeTable	For the purposes of archiving, whether the entity has the semantics of an edge table.	false
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 BillingCenter.	None
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	false
extendable	<i>Internal.</i> If true, it is possible to extend this entity.	true
final	<p>If true, you cannot subtype the entity. If false, you can define subtypes using this entity as the supertype.</p> <p><b>IMPORTANT</b> If you define this incorrectly, BillingCenter generates an error message upon resource verification and the application server refuses to start. BillingCenter generates this verification error:</p> <ul style="list-style-type: none"> <li>• If you attempt to subtype an entity that is marked as final that exists in the metadata folder in Studio.</li> <li>• If you attempt to subtype an entity that is marked as final that exists in the extensions folder in Studio.</li> </ul>	true
generateInternallyIfAbsent	<i>Internal.</i> Do not use.	false
ignoreForEvents	<p>If you change (or add, or remove) an entity X that does not generate events, then BillingCenter searches for all event-generating entity instances that specify X. If BillingCenter 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, BillingCenter 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 BillingCenter ignores that link as it determines what entities specify another entity.</p> <ul style="list-style-type: none"> <li>• At the entity level, the ignoreForEvents attribute means changes and additions or removals from the entity do not cause Changed events to fire for any other entity.</li> <li>• At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events.</li> </ul>	false
instrumentationTable	<i>Internal.</i>	false
loadable	If true, you can load the entity through staging tables.	true

<entity> attribute	Description	Default
lockable	<p><i>Internal.</i> If set to true, BillingCenter adds a lock column (<code>lockingcolumn</code>) to the table for this entity. BillingCenter uses this to acquire an update lock on a row. The most common use is on objects in which it is important to implement safe ordering of messages. In that case, the entity that imposes the safe ordering needs to be lockable.</p> <p><b>IMPORTANT</b> Guidewire strongly recommends that you do not use this locking mechanism.</p>	false
overwrittenInStagingTable	<p><i>Internal.</i> If true and the entity is loadable, the loader process auto-populates the staging table during import.</p> <p><b>IMPORTANT</b> If set to true, do not attempt to populate the table yourself, because the loader import process overwrites this table.</p>	false
platform	<p><i>Internal.</i> Do not use. The only real effect is to change the location in which the table appears in a data distribution report.</p>	false
priority	<p>The priority of the corresponding subtype key. This value is meaningful only for entities participating in a subtype hierarchy, which can be either the &lt;subtype&gt; entities or the root &lt;entity&gt;.</p>	-1
readOnly	<p><i>Optional.</i> The typical use of read-only entities is for tables of reference data that you import as administrative data and then never touch again.</p> <p>You can add a read-only entity only to a bundle that has the <code>allowReadOnlyBeanChanges()</code> flag set on its commit options. That means that inserting, modifying or deleting a read-only entity requires one of these special bundles.</p> <p>You cannot set bundle commit options from Gosu. Therefore, you cannot modify these entities from Gosu, unless some Gosu-accessible interface gives you a special bundle. The administrative XML import tools use such a special bundle. However, Guidewire uses these tools internally only in the PolicyCenter product model.</p>	None
setterScriptability	See “Data Objects and Scriptability” on page 156 for information.	None
subpackage	Subpackage to which the class corresponding to this entity belongs.	None
table	<p><i>Required.</i> The name of the database table in which BillingCenter stores the data for this entity. BillingCenter automatically prefixes table names with <code>bc_</code> for base entities and <code>bcx_</code> for extension entities.</p> <p>Guidewire recommends the following table naming conventions:</p> <ul style="list-style-type: none"> <li>• Do not begin the table name with any product-specific extension.</li> <li>• Use all lower-case letters.</li> <li>• Use letters only.</li> </ul> <p>Guidewire enforces the following restrictions on the maximum allowable length of the table name:</p> <ul style="list-style-type: none"> <li>• <code>loadable="true"</code> — maximum of 25 characters</li> <li>• <code>loadable="false"</code> — maximum of 26 characters</li> </ul>	None
temporary	<p><i>Internal.</i> If true, then this table is a temporary table that BillingCenter uses only during installation or upgrade.</p> <p>BillingCenter deletes all temporary tables after it completes the installation or the upgrade.</p>	false
type	<p><i>Required.</i> See “Overview of Data Entities” on page 149 for a discussion of data entity types.</p>	None

<entity> attribute	Description	Default
typelistTableName	<p>If you create a non-final entity, then ClaimCenter automatically creates a typelist to keep track of the subtypes of that entity. That typelist has an associated database table. If you do not specify a value for this attribute, then ClaimCenter uses the name of the entity as the table name for the subtype typelist.</p>	None
	<p>However, ClaimCenter places a restriction of 25 characters on the length of the database table name. You use this attribute to specify the database table name for the typelist if an entity name is too long to become a valid typelist table name.</p>	
	<p>It is not valid to use this attribute with entity types marked as final.</p>	
validateOnCommit	<p><i>Internal.</i> Do not use. If true, BillingCenter validates this entity during a commit of a bundle that contains this entity.</p>	true

### Subelements of <entity>

The <entity> element contains the following subelements.

<entity> subelement	Description
array	See “<array>” on page 176.
checkconstraint	<i>Internal.</i>
column	See “<column>” on page 178.
customconsistencycheck	<i>Internal.</i>
datetimeordering	<i>Internal.</i>
dbcheckbuilder	<i>Internal.</i>
edgeForeignKey	See “<edgeForeignKey>” on page 182.
events	See “<events>” on page 185.
foreignkey	See “<foreignkey>” on page 186.
fulldescription	See “<fulldescription>” on page 188.
implementsEntity	See “<implementsEntity>” on page 188.
implementsInterface	See “<implementsInterface>” on page 189.
index	See “<index>” on page 190.
jointableconsistencycheck	<i>Internal.</i>
monetaryamount	Handles monetary amounts. The <monetaryamount> subelement is a compound data type that stores its values in two separate database columns: a <money> column type, and a typekey to the Currency typelist.
onetoone	See “<onetoone>” on page 191.
remove-index	See “<remove-index>” on page 193.
searchColumn	See “The <searchColumn> Subelement” on page 165
searchTypekey	Defines a search denormalization typekey in the database. The denormalization copies the value of a column on another table into a typekey field on the denormalizing table. You must link the tables through a foreign key. The purpose of this denormalization is to avoid costly joins in performance critical searches.
tableAugmenter	<i>Internal.</i>
typekey	See “<typekey>” on page 194.
validatetypekeyinset	<i>Internal.</i>
validatetypekeynotinset	<i>Internal.</i>

### The <searchColumn> Subelement

The <searchColumn> subelement on <entity> defines a search denormalization column in the database. The denormalization copies the value of a column on another table into a column on the denormalizing table. You must link the tables through a foreign key. The purpose of this denormalization is to avoid costly joins in performance-critical searches.

The use of search denormalization columns adds overhead to updates, as does any denormalization. Guidewire recommends that you only use these columns if there is an identifiable performance problem with a search that is directly related to the join between the two tables.

**Note:** It is possible to have a <searchColumn> sublement on the <extension> and <subtype> elements as well.

The <searchColumn> element contains the following attributes.

<searchColumn> attribute	Description	Default
columnName	Name to use for the database column corresponding to this property. If you do not specify a value, then BillingCenter uses the name value instead.	None
deprecated	If true, then BillingCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.  If you deprecate an item, use the description to explain why.  For more information, see “The deprecated Attribute” on page 175.	false
desc	Description of the intended purpose of this column.	None
name	<i>Required.</i> Name of the column on the table <i>and</i> the field on the entity. The name value maps to the accessor and mutator methods of a field on the entity, not the actual private member field. For example, name maps to setName and getName, not the private _name member field.  Column names must contain letters only. A column name cannot contain an underscore.	None
sourceColumn	<i>Required.</i> Name of the column on the source entity, whose value this column copies. The sourceColumn must not name a localized column.	None
sourceForeignKey	<i>Required.</i> Name of a foreign key field on this entity, which refers to the source entity for this search denormalization column. The sourceForeignKey must not be importable against existing objects.	None
sourceSubtype	Optional name of the particular subtype on which the source column is defined. If not specified, then BillingCenter assumes that the source column to exist on the entity referred to by the source object. However, you must specify this value if the sourceColumn is on a subtype of the entity referred to by sourceForeignKey.	None

For example, suppose that you set the following attribute definitions:

- `searchColumn – MyDenormColumn`
- `sourceForeignKey – Source`
- `sourceColumn – SourceField`

This declaration says:

Copy the value of *SourceField* on the object pointed to by the foreign key named *Source* into the field named *MyDenormColumn*. BillingCenter automatically populates the column as part of bundle commit, staging table load, and database upgrade.

If you need to denormalize a field on a subtype of the entity referred to by the foreign key, then you can specify the optional `sourceSubtype` attribute.

As with linguistic denormalization columns, you cannot access the value of these search denormalization columns in memory. The value is only available in the database. Thus, you can only access the value through a database query.

It is possible to make a query against a search denormalization column that is a denormalization of a linguistic denormalization column. In that case, the query generator knows not to wrap the column values in the linguistic denormalization function. This preserves the optimization that linguistic denormalization columns provide.

It is important to understand that search denormalization columns specify one column only — the column that you specify with the `sourceColumn` attribute. So, if you want to denormalize both a column and its linguistic denormalization, then you need two separate search denormalization columns. However, in this case, you typically would just want to denormalize the linguistic denormalization column. You would only want to denormalize the source column if you wanted to support case-sensitive searches on it.

Search denormalization columns can only specify `<column>` or `<typekey>` fields.

## Extension Data Objects

An extension data object is the standard data object that you use to extend an already existing data object or entity. Guidewire defines this object in the data model metadata files as the `<extension>` XML root element.

### See also

- For information on how to extend the base data objects, see “Modifying the Base Data Model” on page 205.

### Attributes of `<extension>`

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

<code>&lt;extension&gt;</code> attribute	Description	Default
<code>entityName</code>	<i>Required.</i> This value must match the file name of the entity that it extends. BillingCenter generates an error at resource verification if the value set that you set for the <code>entityName</code> attribute for an extension does not match the file name.	None

### Subelements of `<extension>`

The `<extension>` element contains the following subelements.

<code>&lt;extension&gt;</code> subelement	Description
<code>array</code>	See “ <code>&lt;array&gt;</code> ” on page 176.
<code>array-override</code>	Use to override, or flip, the value of the <code>triggersValidation</code> attribute of an <code>&lt;array&gt;</code> element definition on a base data object. See “Working with Attribute Overrides” on page 210 for details.
<code>column</code>	See “ <code>&lt;column&gt;</code> ” on page 178.
<code>column-override</code>	Use to override certain very specific attributes of a base data object. See “Working with Attribute Overrides” on page 210 for details.
<code>description</code>	A description of the purpose and use of the entity.
<code>edgeForeignKey</code>	See “ <code>&lt;edgeForeignKey&gt;</code> ” on page 182.
<code>edgeForeignKey-override</code>	Use to override certain very specific attributes of a base data object.
<code>events</code>	See “ <code>&lt;events&gt;</code> ” on page 185.
<code>foreignkey</code>	See “ <code>&lt;foreignkey&gt;</code> ” on page 186.
<code>foreignkey-override</code>	Use to override, or flip, the value of the <code>triggersValidation</code> attribute of a <code>&lt;foreignkey&gt;</code> element definition on a base data object. See “Working with Attribute Overrides” on page 210 for details.
<code>implementsEntity</code>	See “ <code>&lt;implementsEntity&gt;</code> ” on page 188.

<extension> subelement	Description
implementsInterface	See “<implementsInterface>” on page 189.
index	See “<index>” on page 190.
internalonlyfields	<i>Internal.</i>
monetaryamount	Handles monetary amounts. The <monetaryamount> subelement is a compound data type that stores its values in two separate database columns: a <money> column type, and a typekey to the Currency typelist.
onetoone	See “<onetoone>” on page 191.
onetoone-override	Use to override, or flip, the value of the triggersValidation attribute of an <onetoone> element definition on a base data object. See “Working with Attribute Overrides” on page 210 for details.
remove-index	See “<remove-index>” on page 193.
searchColumn	See “The <searchColumn> Subelement” on page 165
searchTypekey	Defines a search denormalization typekey in the database. The denormalization copies the value of a column on another table into a typekey field on the denormalizing table. You must link the tables through a foreign key. The purpose of this denormalization is to avoid costly joins in performance critical searches.
typekey	See “<typekey>” on page 194.
typekey-override	Use to override certain specific attributes, or fields, of a <typekey> element definition on a base data object. See “Working with Attribute Overrides” on page 210 for details.

## Non-persistent Entity Data Objects

A non-persistent entity data object defines a temporary entity that BillingCenter creates and uses only during the time that the BillingCenter server is running. If the server shuts down, BillingCenter discards the entity data. It is not possible to commit a non-persistent entity object to the database.

Guidewire defines this object in the data model metadata files as the <nonPersistentEntity> XML root element.

**Note:** You cannot extend a persistent entity with a non-persistent entity.

### Guidewire Recommendations for Non-persistent Entities

Guidewire recommends that you do not create or extend non-persistent entities as a general rule. In general, do not use non-persistent entities to obtain some desired behavior. A major issue with non-persistent entities is that they do not interact well with data bundles. Passing a non-persistent entity to a PCF page, for example, is generally a bad idea because it generally does not work in the manner that you expect.

The non-persistent 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 non-persistent entity from one context to another, the entity loses any nested arrays or links associated with it. Thus, it is possible to lose parts of the entity graph as the non-persistent entity moves around. Entity serialization is also less efficient and less controllable than using a custom class that contains only the data that it really needs.

Guidewire recommends, therefore, that you use a Gosu class in situations in which you want the behavior of a non-persistent entity. For example:

- If you want the behavior of a non-persistent entity in web services, do not use a non-persistent entity. Instead, Guidewire recommends that you create a Gosu class and then expose that as a web service rather than relying on non-persistent entities and entity serialization.
- If you want a field that behaves, for example, as nonnegativeinteger column, do not use a non-persistent 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 227 for information on how to associate data types with object properties using the annotation syntax.

### Attributes of <nonPersistentEntity>

The <nonPersistentEntity> element contains the following attributes.

<nonPersistentEntity> attribute	Description	Default
abstract	If true, you cannot create an instance of the entity type at runtime. Instead, you must declare a subtype entity with abstract=false, which you can instantiate. Any of the generated code is abstract.	false
desc	A description of the purpose and use of the entity.	None
displayName	<i>Optional.</i> Creates a more human-readable form of the entity name. You can access this name using the following:  <code>entity.DisplayName</code>  If you do not specify a value for the DisplayName attribute, then the <code>entity.DisplayName</code> method returns the value of the entity attribute, instead. If you subtype an entity that has a specified display name, then the <code>entity.DisplayName</code> method returns the name of the subtype key.	None
entity	<i>Required.</i> The name of the entity. You use this name to access the entity in data views, rules, and other areas within BillingCenter.	None
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	false
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
platform	<i>Internal.</i> Do not use. The only real effect is to change the location in which the table appears in a data distribution report.	false
priority	The priority of the corresponding subtype key. This value is meaningful only for entities participating in a subtype hierarchy, which can be either the <subtype> entities or the root <entity>.	-1
subpackage	Subpackage to which the class corresponding to this entity belongs.	None
typelistTableName	If you create a non-final entity, then ClaimCenter automatically creates a typelist to keep track of the subtypes of that entity. That typelist has an associated database table. If you do not specify a value for this attribute, then ClaimCenter uses the name of the entity as the table name for the subtype typelist.  However, ClaimCenter places a restriction of 25 characters on the length of the database table name. You use this attribute to specify the database table name for the typelist if an entity name is too long to become a valid typelist table name.  It is not valid to use this attribute with entity types marked as final.	None

### Subelements of <nonPersistentEntity>

The <nonPersistentEntity> element contains the following subelements.

<nonPersistentEntity> subelement	Description
array	See “<array>” on page 176.
column	See “<column>” on page 178.
edgeForeignKey	See “<edgeForeignKey>” on page 182.
foreignkey	See “<foreignkey>” on page 186.
fulldescription	See “<fulldescription>” on page 188.
implementsEntity	See “<implementsEntity>” on page 188.
implementsInterface	See “<implementsInterface>” on page 189.

<nonPersistentEntity> subelement	Description
monetaryamount	Handles monetary amounts. The <monetaryamount> subelement is a compound data type that stores its values in two separate database columns: a <money> column type, and a typekey to the Currency typelist.
onetoone	See “<onetoone>” on page 191.
typekey	See “<typekey>” on page 194.

## Subtype Data Objects

A subtype 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.

BillingCenter does not associate a separate database table with a subtype. Instead, BillingCenter 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, BillingCenter 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 BillingCenter metadata service enforces this for all data operations.

You can only define a subtype for any entity that has its `final` attribute set to `false`. BillingCenter automatically creates a `Subtype` field for non-final entities.

### Attributes of <subtype>

The <subtype> element contains the following attributes:

<subtype> attribute	Description	Default
abstract	If true, you cannot create an instance of the entity type at runtime. Instead, you must declare a subtype entity with <code>abstract=false</code> , which you can instantiate. Any of the generated code is abstract.	false
desc	A description of the purpose and use of the subtype.	None
displayName	<i>Optional.</i> Occasionally in the BillingCenter interface, you want to display the subtype name of subtyped entity instances. Use the <code>displayName</code> attribute to specify a String to display as the subtype name. You can access this name using the following:  <code>entity.DisplayName</code>  If you do not specify a value for the <code>displayName</code> attribute, then BillingCenter displays the name of the entity. The entity name is often not user-friendly. For a description of the <code>displayName</code> attribute, see “Entity Data Objects” on page 161.	None
entity	<i>Required.</i>	
final	If true, you cannot subtype the entity. If false, you can define subtypes using this entity as the supertype.	false
platform	<i>Internal.</i> Do not use. The only real effect is to change the location in which the table appears in a data distribution report.	false
priority	The priority of the corresponding subtype key. This value is meaningful only for entities participating in a subtype hierarchy, which can be either the <subtype> entities or the root <entity>.	-1

<subtype> attribute	Description	Default
readOnly	<p><i>Optional.</i> The typical use of read-only entities is for tables of reference data that you import as administrative data and then never touch again.</p>	None
	<p>You can add a read-only entity only to a bundle that has the <code>allowReadOnlyBeanChanges()</code> flag set on its commit options. That means that inserting, modifying or deleting a read-only entity requires one of these special bundles.</p>	
	<p>You cannot set bundle commit options from Gosu. Therefore, you cannot modify these entities from Gosu, unless some Gosu-accessible interface gives you a special bundle. The administrative XML import tools use such a special bundle. However, Guidewire uses these tools internally only in the PolicyCenter product model.</p>	
setterScriptability	See “Data Objects and Scriptability” on page 156 for information.	None
subpackage	Subpackage to which the class corresponding to this entity belongs.	None
supertype	<i>Required.</i>	

### Subelements of <subtype>

The <subtype> element contains the following subelements.

<subtype> subelement	Description
array	See “<array>” on page 176.
checkconstraint	<i>Internal.</i>
column	See “<column>” on page 178.
customconsistencycheck	<i>Internal.</i>
datetimeordering	<i>Internal.</i>
dbcheckbuilder	<i>Internal.</i>
edgeForeignKey	See “<edgeForeignKey>” on page 182.
events	See “<events>” on page 185.
foreignkey	See “<foreignkey>” on page 186.
fulldescription	See “<fulldescription>” on page 188.
implementsEntity	See “<implementsEntity>” on page 188.
implementsInterface	See “<implementsInterface>” on page 189.
index	See “<index>” on page 190.
jointableconsistencycheck	<i>Internal.</i>
monetaryamount	Handles monetary amounts. The <monetaryamount> subelement is a compound data type that stores its values in two separate database columns: a <money> column type, and a typekey to the Currency typelist.
onetoone	See “<onetoone>” on page 191.
searchColumn	See “The <searchColumn> Subelement” on page 165
searchTypekey	Defines a search denormalization typekey in the database. The denormalization copies the value of a column on another table into a typekey field on the denormalizing table. You must link the tables through a foreign key. The purpose of this denormalization is to avoid costly joins in performance critical searches.
tableAugmenter	<i>Internal.</i>
typekey	See “<typekey>” on page 194.
validatetypekeyinset	<i>Internal.</i>
validatetypekeynotinset	<i>Internal.</i>

## Subtypes and Typelists

After you define a new subtype, BillingCenter automatically adds that entity type to the associated entity typelist. This is true, even if BillingCenter marks that typelist as `final`.

For example, suppose that you define an `Inspector` entity as a subtype of `Person`.

```
<?xml version="1.0"?>
<subtype xmlns="http://guidewire.com/datamodel" desc="Professional inspector" displayName="Inspector"
    entity="InspectorExt"
    supertype="Person">
    <column name="InspectorLicenseExt" type="varchar" desc="Inspector's business license number">
        <columnParam name="size" value="30"/>
    </column>
</subtype>
```

Notice that while `InspectorExt` is subtype of `Person`, `Person`, itself, is a subtype of `Contact`. BillingCenter automatically adds the new `InspectorExt` type to the `Contact` typelist. This is true, even though BillingCenter marks the `Contact` typelist as `final`.

To see this change:

- In the *BillingCenter Data Dictionary*, you must restart the application server.
- In the `Contact` typelist in Studio, you must restart Studio.

### See also

- “Defining a Subtype” on page 215

## View Entity Data Objects

A view entity is a logical view of entity data. You can use a view entity to enhance performance during the viewing of tabular data. A view entity provides a logical view of data for an entity of interest to a `ListView`. A view entity can include paths from the root or primary entity to other related entities. For example, the `Activity` entity has a corresponding view entity called `ActivityView`.

Unlike a standard entity, a view entity type does not have an underlying database table. BillingCenter does not persist view entities to the database. Instead of storing data, a view entity restricts the amount of data that a database query returns. A view entity does not represent or create a *materialized view*, which is a database table that caches the results of a database query.

Queries against a view entity type are actually run against the normal entity table in the database, as specified by the `primaryEntity` attribute of the view entity definition. The query against a view entity automatically adds any joins necessary to retrieve view entity columns if they include a bean path. However, access to view entity columns is not possible when constructing the query.

A view entity improves the performance of BillingCenter on frequently used pages that list entities. Like other entities, you can subtype a view entity. Because BillingCenter can export view entity types, it generates SOAP interfaces for them.

**Note:** If you create or extend a view entity that references a column that is of `type="currencyamount"`, then you must handle the view entity extension in a particular manner. See “Extending an Existing View Entity with a Currency Column” on page 219 for details.

Guidewire defines this object in the data model metadata files as the `<viewEntity>` XML root element.

### Attributes of <viewEntity>

The <viewEntity> element contains the following attributes:

<viewEntity> attribute	Description	Default
abstract	If true, you cannot create an instance of the entity type at runtime. Instead, you must declare a subtype entity with abstract=false, which you can instantiate. Any of the generated code is abstract.	false
desc	A description of the purpose and use of the entity.	None
displayName	<i>Optional.</i> Creates a more human-readable form of the entity name. You can access this name using the following:  viewEntity.DisplayName  If you do not specify a value for the DisplayName attribute, then the entity.DisplayName method returns the value of the entity attribute, instead. If you subtype an entity that has a specified display name, then the entity.DisplayName method returns the name of the subtype key.	None
entity	<i>Required.</i> Name of this viewEntity object.	None
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
extendable	If true, it is possible to extend this entity.	true
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.	true
platform	<i>Internal.</i> Do not use. The only real effect is to change the location in which the table appears in a data distribution report.	false
primaryEntity	<i>Required.</i> The primary entity type for this viewEntity object. The primary entity must be keyable. See “Data Entities and the Application Database” on page 153 for information on keyable entities.	None
priority	For supertypes and subtypes, the priority of the corresponding subtype key.	-1
showRetiredBeans	Whether to show retired beans in the view.	None
subpackage	Subpackage to which the class corresponding to this entity belongs.	None
supertypeEntity	<i>Optional.</i> The name of supertype of this entity.	None
typelistTableName	If you create a non-final entity, then ClaimCenter automatically creates a typelist to keep track of the subtypes of that entity. That typelist has an associated database table. If you do not specify a value for this attribute, then ClaimCenter uses the name of the entity as the table name for the subtype typelist.  However, ClaimCenter places a restriction of 25 characters on the length of the database table name. You use this attribute to specify an alternate database table name for the typelist if an entity name is too long to become a valid typelist table name.  It is not valid to use this attribute with entity types marked as final.	None

### Subelements of <viewEntity>

The <viewEntity> elements contain the following subelements:

<viewEntity> subelement	Description
computedcolumn	Specifies a column with row values that Guidewire computes while querying the database. For example, the values of a computed column might be the sum of the values from two database columns (col1 + col2).
computedtypekey	Specifies a typekey that has some type of transformation applied to it during querying from the database.
fulldescription	See the discussion following the table.
viewEntityColumn	Represents a column in a viewEntity table
viewEntityLink	Uses to access another entity through a foreign key. Typically, you use this value within the BillingCenter interface to create a link to that entity.

<viewEntity> subelement	Description
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 BillingCenter interface.
viewEntityTypekey	Represents a typekey column in a viewEntity table.

The *Data Dictionary* uses the `fulldescription` subelement. The following example illustrates how to use this element:

```
<fulldescription>
  <![CDATA[<p>Aggregates the information needed to display one activity row (base entity for all other
  activity views).</p>]]>
</fulldescription>
```

The other subelements all require both a name and path attribute. The following code illustrates this:

```
<viewEntityName name="RelActAssignedUserName" path="RelatedActivity.AssignedUser"/>
```

Specify the path value relative to the `primaryEntity` on which you base the view.

The `computedcolumn` takes a required `expression` attribute and an additional, optional `function` attribute. The following is an example of a `computedcolumn`:

```
<computedcolumn name="Amount" expression="${1}" paths="LineItems.Amount" function="SUM"/>
```

The expression for this column can take multiple column values `${column_num}` passed from the BillingCenter interface. For example, a valid expression is: `${1} - ${2}` with `${1}` the first column and `${2}` the second column. The `function` value must be an SQL function that you can apply to this expression. The following are legal values:

- SUM
- AVG
- COUNT
- MIN
- MAX

**Note:** If the SQL function aggregates data, BillingCenter applies an SQL group automatically.

## View Entity Extension Data Objects

You use the view entity extension entity to extend the definition of a `viewEntity` entity. Guidewire defines this object in the data model metadata files as the `<viewEntityExtension>` XML root element.

### Attributes of `<viewEntityExtension>`

The `<viewEntityExtension>` element contains the following attributes:

<viewEntityExtension> attribute	Description	Default
entityName	<p><i>Required.</i> This value must match the file name of the <code>viewEntityExtension</code> that it extends.</p> <p>BillingCenter generates an error at resource verification if the value set that you set for the <code>entityName</code> attribute for a <code>viewEntityExtension</code> does not match the file name.</p>	None

### Subelements of <viewEntityExtension>

The <viewEntityExtension> element contains the following subelements:

<viewEntityExtension> subelement	Description
computedcolumn	Specifies a column with row values that Guidewire computes while querying the database. For example, the values of a computed column might be the sum of the values from two database columns ( <code>col1 + col2</code> ).
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. The <code>viewEntityColumn</code> element contains a <code>path</code> attribute that you use to define the entity path for the column: <ul style="list-style-type: none"> <li>• The <code>path</code> attribute definition cannot traverse arrays.</li> <li>• The <code>path</code> attribute is always relative to the primary entity on which you base the view.</li> </ul> <p><b>Note:</b> If you reference a column of type <code>currencyamount</code>, you must also define the <code>currencyProperty</code> specified in the original column definition on the <code>viewEntity</code> entity. See “Extending an Existing View Entity” on page 218 for an example of this.</p>
viewEntityLink	Uses to access another entity through a foreign key. Typically, you use this value within the BillingCenter 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 BillingCenter interface.
viewEntityTypekey	Represents a typekey column in a <code>viewEntity</code> table.

#### Important Caution

Guidewire strongly recommends that you not create a view entity extension—`viewEntityExtension`—that causes traversals into revisioned (`effdated`) data. Doing so has the possibility of returning duplicate rows if any revisioning in the traversal path splits an entity.

Instead, try one of the following:

- Denormalize the desired data onto a non-`effdated` entity.
- Add domain methods to the implementation of the `View` entity.

## Data Object Subelements

This topic describes the subelements that you can use in metadata definition files. These subelements are:

- `<array>`
- `<column>`
- `<edgeForeignKey>`
- `<events>`
- `<foreignkey>`
- `<fulldescription>`
- `<implementsEntity>`
- `<implementsInterface>`
- `<index>`
- `<onetoone>`
- `<remove-index>`

- <tag>
- <typekey>

### Subelements for Internal Use Only

Do not use the following entity subelements. Guidewire uses these subelements for internal purposes only.

- <aspect>
- <checkconstraint>
- <customconsistencycheck>
- <datetimeordering>
- <dbcheckbuilder>
- <jointableconsistencycheck>
- <tableAugmenter>
- <validatetypekeyinset>
- <validatetypekeynotinset>

### The deprecated Attribute

The `deprecated` attribute applies to the following subelements:

- <array>
- <column>
- <componentref>
- <edgeForeignKey>
- <foreignkey>
- <onetoono>
- <searchColumn>
- <typekey>

The `deprecated="true"` attribute does not alter the database in any way. Instead, the `deprecated` attribute marks a data field as deprecated in the *Data Dictionary* and places a `Deprecated` annotation on the field in the *Guidewire Studio API Reference*. The `deprecated` attribute supports organizations that want to remove a field in a two-phase process.

In the first phase, you add the `deprecated` attribute to the field subelement. Studio indicates the field is deprecated whenever Gosu code references the field. During this first phase, developers work to remove the deprecated field from their code. The second phase occurs after developers remove all occurrences of the deprecated field.

In the second phase, you drop the field from the entity definition. In some cases, Guidewire will drop the column from the database automatically to synchronize the physical database with your revised data model. In most cases however, the DBA must alter the database with SQL statements run against the database to synchronize the database with your revised data model.

Guidewire generally recommends against using the `deprecated` attribute and the two-phase removal process. If you deprecate a field, Studio signals to the development team that the field is no longer used. The DBA does not receive this information. Over time, with a number of deprecated fields, the DBA manages an ever larger amount of unused information in the physical database. To avoid managing unused data, Guidewire strongly recommends that you keep your physical database and the data model of your application synchronized by dropping unused fields instead of deprecating them.

## <array>

An array defines a set of additional entities of the same type to associate with the main entity. For example, a Account entity includes an array of Document entities.

### Attributes of <array>

The <array> element contains the following attributes:

<array> attribute	Description	Default
arrayentity	Required. The name of the entity that makes up the array.	None
arrayfield	Optional. Name of the field in the array table that is the foreign key back to this table. However, you do not need to define a value if the array entity has exactly one foreign key back to this entity.  Note that even if you define only one foreign key explicitly, additional foreign keys may be created implicitly. For example, CreateUserID is automatically added to an editable entity. In that case, arrayfield would be required because there is more than one foreign key.	None
cascadeDelete	If true, then BillingCenter deletes the array elements also if you delete the array container.	false
deprecated	If true, then BillingCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.  If you deprecate an item, use the description to explain why.  For more information, see “The deprecated Attribute” on page 175.	false
desc	A description of the purpose and use of the array.	None
exportable	Deprecated. Only used with RPCE web services, which are deprecated.	true
getterScriptability	See “Data Objects and Scriptability” on page 156 for information.	all
ignoreforevents	If you change (or add, or remove) an entity X that does not generate events, then BillingCenter searches for all event-generating entity instances that specify X. If BillingCenter 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, BillingCenter 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 BillingCenter ignores that link as it determines what entities specify another entity. <ul style="list-style-type: none"> <li>• 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.</li> <li>• At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events.</li> </ul>	false
name	Required. The name of the property corresponding to this array	None
owner	If true, this entity owns the objects in the array. <ul style="list-style-type: none"> <li>• If you delete the owning object, then BillingCenter deletes the array items as well.</li> <li>• If you update the contents of the array, then BillingCenter considers the owner as updated as well.</li> </ul>	false
requiredmatch	One of the following values <ul style="list-style-type: none"> <li>• all – There must be at least one matching row in the array for every row from this table. For example, there must be at least one check payee for every check.</li> <li>• none – There is no requirement for matching rows.</li> <li>• nonretired – There must be at least one matching row for every non-retired row from this table.</li> </ul>	None

<array> attribute	Description	Default
setterScriptability	See “Data Objects and Scriptability” on page 156 for information.	all
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"> <li>• The addition of an object to the array</li> <li>• The removal of an object from the array</li> <li>• The modification of an object in the array</li> </ul>	false
See the discussion on this attribute that follows this table.		

If set to `true`, the `triggersValidation` attribute can trigger additional BillingCenter processing. Exactly what happens depends on several different factors:

- If the parent entity for the array is validatable, then any modification to the array triggers the execution of the Preupdate and Validation rules on the parent entity. Validation occurs whenever BillingCenter attempts to commit a bundle that contains the parent entity. For an entity to be validatable, it must implement the `Validatable` delegate.
- If the parent entity has preupdate rules, but no validation rules, then BillingCenter executes the preupdate rules on the commit bundle. This is the case only if configuration parameter `UseOldStylePreUpdate` is set to `true`, which is the default. If `UseOldStylePreUpdate` is set to `false`, BillingCenter invokes the `IPreUpdateHandler` plugin on the commit bundle instead. Then, BillingCenter executes the logic defined in the plugin on the commit bundle.
- If the parent entity has validation rules, but no preupdate rules, then BillingCenter executes the validation rules on the commit bundle.
- If the parent entity has neither preupdate nor validation rules then the following occurs:
  - a. In the case of `UseOldStylePreUpdate=true`, BillingCenter does nothing.
  - b. In the case of `UseOldStylePreUpdate=false`, BillingCenter calls the `IPreUpdateHandler` plugin on the commit bundle.

#### Subelements of <array>

The `<array>` element contains the following subelements:

<array> subelement	Description
array-association	This subelement contains the following attributes: <ul style="list-style-type: none"> <li>• <code>hasContains</code> (default = <code>false</code>)</li> <li>• <code>hasGetter</code> (default = <code>true</code>)</li> <li>• <code>hasSetter</code> (default = <code>false</code>)</li> <li>• <code>valueField</code> (default = <code>ID</code>)</li> </ul> It also contains the following subelements of its own, each of which can exist, at most, one time: <ul style="list-style-type: none"> <li>• <code>constant-map</code></li> <li>• <code>subtype-map</code></li> <li>• <code>typelist-map</code></li> </ul> See “Typelist Mapping Associative Arrays” on page 201 for more information.
fulldescription	See “<fulldescription>” on page 188.

<array> subelement	Description
link-association	<p>This subelement contains the following attributes:</p> <ul style="list-style-type: none"> <li>• hasGetter (default = true)</li> <li>• hasSetter (default = false)</li> <li>• valueField (default = ID)</li> </ul> <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"> <li>• constant-map</li> <li>• subtype-map</li> <li>• typeList-map</li> </ul> <p>See “Subtype Mapping Associative Arrays” on page 199 for more information.</p>
tag	See “<tag>” on page 194.

## <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 210 for details.

### Attributes of <column>

The <column> element contains the following attributes:

<column> attribute	Description	Default
autoincrement	The name of a database sequence used as the source of values for the column. This attribute is applicable only for integer columns, and only for implementations where BillingCenter relies on the database for the sequence. Do not use the same sequence in more than one autoincrement column. There can be at most one autoincrement column per table.	None
columnName	<p><i>Optional.</i> If specified, BillingCenter uses this value as the column name of the corresponding database column. If you do not specify a columnName value, then BillingCenter uses the value of the name attribute for the database column name. The maximum length for a column name is 30 characters.</p> <p><b>IMPORTANT</b> All column names on a table must be unique within that table. Otherwise, Studio displays an error if you verify the resource and the application server fails to start.</p>	None
createhistogram	<p>Whether to create a histogram on the column during an update to the database statistics.</p> <p><b>Note:</b> It is possible to override this attribute on an existing column in an extension (*.etx) file using the &lt;column-override&gt; element. You can use the override to turn off an existing histogram or to create one that did not previously exist.</p> <p>This change does not take effect during an upgrade. The change occurs <i>only</i> if you regenerate statistics for the affected table by using the Guidewire maintenance_tools command.</p>	false
see also	<ul style="list-style-type: none"> <li>• “Working with Attribute Overrides” on page 210</li> <li>• “Configuring Database Statistics” on page 39 in the <i>System Administration Guide</i></li> <li>• “Maintenance Tools Command” on page 188 in the <i>System Administration Guide</i></li> </ul>	
default	Default value given to the field during new entity creation.	None

<column> attribute	Description	Default
deprecated	<p>If true, then BillingCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.</p>	false
	<p>If you deprecate an item, use the description to explain why.</p>	
	<p>For more information, see “The deprecated Attribute” on page 175.</p>	
desc	<p>A description of the purpose and use of the field.</p>	None
exportable	<p><i>Deprecated.</i> Only used with RPCE web services, which are deprecated.</p>	true
getterScriptability	<p>See “Data Objects and Scriptability” on page 156 for information.</p>	all
ignoreforevents	<p>If you change (or add, or remove) an entity X that does not generate events, then BillingCenter searches for all event-generating entity instances that specify X. If BillingCenter finds any of these event-generating entity instances, it generates Changed events for those entity instances.</p>	false
	<p>To determine what entities reference a non-event-generating entity, BillingCenter 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 BillingCenter ignores that link as it determines what entities specify another entity.</p>	
	<ul style="list-style-type: none"> <li>• 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.</li> <li>• At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events.</li> </ul>	
loadable	<p>If true, you can load the field through staging tables. A staging table can contain a column mapping to the field.</p>	true
LoadedByCallback	<p><i>Internal.</i> If true, then the loading code does not use a default value or report a warning if the column is nullable without a default.</p>	false
name	<p><i>Required.</i> The name of the column on the table and the field, or property, on the entity. BillingCenter uses this value as the column name <i>unless</i> you specify a columnName attribute. Use this name to access the column in data views, rules, and other areas within BillingCenter.</p>	None
	<p><b>IMPORTANT</b> All column names on a table must be unique within that table. Otherwise, Studio displays an error if you verify the resource and the application server fails to start.</p>	
nullok	<p>Whether the column can contain null values.</p>	true
	<p>In general, this attribute is always true, as many tables include columns that do not require a value at different points in the process.</p>	
overwrittenInStagingTable	<p><i>Internal.</i> If true and the entity is loadable, the loader process auto-populates the staging table during import.</p>	false
	<p><b>IMPORTANT</b> If set to true, do not attempt to populate the table yourself as the loader import process overwrites this table.</p>	
scalable	<p>Whether this value scales as the effective and expired dates change. This attribute applies only to number-type values. For example, you cannot scale a varchar. Also, it only applies to effective dated types.</p>	false
setterScriptability	<p>See “Data Objects and Scriptability” on page 156 for information.</p>	all
soapnullok	<p><i>Deprecated.</i> Only used with RPCE web services, which are deprecated.</p>	None

<column> attribute	Description	Default
supportsLinguisticSearch	<p>Applies only to columns of varchar-based data types.</p> <ul style="list-style-type: none"> <li>If true, searches performed on this field are linguistic.</li> <li>If false, searches are binary.</li> </ul>	false
type	<p><i>Required.</i> Data type of the column, or field. In the base configuration, Guidewire defines a number of data types and stores their metadata definition files (*.dti) in modules/configuration/config/datatypes.</p> <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 &lt;column&gt; 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 BillingCenter 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. BillingCenter 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"> <li>BillingCenter represents a column whose type is datetime in memory as instances of java.util.Date.</li> <li>BillingCenter stores this type of value in the database as TIMESTAMP.</li> </ul>	None

### Subelements of <column>

The <column> element contains the following subelements:

#### <columnParam> Subelement

<column> subelement	Description	Default
columnParam	See “<columnParam> Subelement” on page 180.	None
fulldescription	See “<fulldescription>” on page 188.	None
Localization	See “<localization> Subelement” on page 182.	None
tag	See “<tag>” on page 194.	None

You use the <columnParam> element to set parameters that a column type requires. The type attribute of a column determines which parameters you can set or modify by using the <columnParam> subelement. You can determine the list of parameters that a column type supports by looking up the type definition in its .dti file.

For example, if you have a mediumtext column, you can determine the valid parameters for that column by examining file mediumtext.dti. This file indicates that you can modify the following attributes of a mediumtext column:

- encryption
- logicalSize
- trimwhitespace
- validator

Because you cannot modify the base configuration data type declaration files, you cannot see these files in Guidewire Studio. To view these files, navigate to the directory `modules/configuration/config/datatypes`.

The following example, from `Account.eti` in PolicyCenter, illustrates how to use this subelement to define certain column parameters.

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel"
    desc="An account is ..."
    entity="Account"
    ...
    table="account"
    type="retireable">
    ...
    <column desc="Business and Operations Description."
        name="BusOpsDesc"
        type="varchar">
        <columnParam name="size" value="240"/>
    </column>
    ...
</extension>
```

#### Parameters that You Can Define by Using `<columnParam>`

The following list describes the parameters that you can define by using `<columnParam>`, depending on which parameters are listed as valid in the `.dti` file of the data type.

Parameter	Description
<code>countryProperty</code>	Name of a property on the owning entity that returns the country to use for localizing the data format for this column.
<code>currencyProperty</code>	Name of a property on the owning entity that returns the currency for this column.
<code>encryption</code>	Whether BillingCenter stores this column in encrypted format. This only applies to text-based columns.  Guidewire allows indexes on encrypted columns, or fields. However, because Guidewire stores encrypted fields as encrypted in the database, you must encrypt the input string and search for an exact match to it.
<code>exchangeRateProperty</code>	Name of a property on the owning entity that returns the exchange rate to use during currency conversions.
<code>extensionProperty</code>	The name of a property on the owning entity whose value contains the phone extension.
<code>logicalSize</code>	The size of this field in the BillingCenter interface. You can use this value for String columns that do not have a maximum size in the database, such as CLOB objects. If you specify a value for the <code>size</code> parameter, then the <code>logicalSize</code> value must be less than or equal to the value of that parameter.
<code>phonecountrycodeProperty</code>	The name of a property on the owning entity whose value contains the country with which to validate and format values.
<code>precision</code>	The <i>precision</i> of the field. Precision is the total number of digits in the number. The <code>precision</code> parameter applies only if the data type of the field allows a precision attribute.
<code>scale</code>	The <i>scale</i> of the field. Scale is the number of digits to the right of the decimal point. The <code>scale</code> parameter applies only if the data type of the field allows a scale attribute.
<code>secondaryAmountProperty</code>	Name of a property on the owning entity that returns the secondary amount related to this currency amount column.
<code>size</code>	Integer size value for columns of type TEXT and VARCHAR. Use these with only column types. This parameter specifies the maximum number of characters, not bytes, that the column can hold.  <b>WARNING</b> The database upgrade utility automatically detects definitions that lengthen or shorten a column. For shortened columns, the utility assumes that you wrote a version check or otherwise verified that the change does not truncate existing column data. For both Oracle and SQL Server, if shortening a column causes the truncation of data, the ALTER TABLE statement in the database fails and the upgrade utility fails.

Parameter	Description
trimwhitespace	Applies to text-based data types. If true, then BillingCenter automatically removes leading and trailing white space from the data value.
validator	The name of a ValidatorDef in fieldvalidators.xml. See “<ValidatorDef>” on page 241.

**See also**

- See “Overriding Data Type Attributes” on page 211 for an example of using a nested <columnParam> subelement within a <column-override> element to set the encryption attribute on a column.

**<localization> Subelement**

For a discussion of the column <localization> element with examples on how to use it, see “Localized Columns in Entities” on page 67 in the *Globalization Guide*.

**Subelements of <localization>**

The <localization> element contains the following subelements:

<localization> subelement	Description	Default
extractable	Whether the localized data entity is marked as Extractable for archiving.	false
nullok	Required. Whether null values are allowed.	None
overlapTable	Whether the localized data entity is marked as OverlapTable for archiving.	false
tableName	The table name of the localized data table.	None
unique	Required. Whether values must be unique.	false

**<edgeForeignKey>**

You use the <edgeForeignKey> element to define a reference to another entity, in a manner similar to the <foreignkey> element. However, you use an edge foreign key in place of a standard foreign key to break a cycle of foreign keys in the data model. Guidewire defines this element in the data model metadata files as the <edgeForeignKey> XML subelement.

**The Data Model and Circular References**

A chain of foreign keys can form a cycle, also known as a *circular reference*, in the data model. As an example of a circular reference, entity type A has a foreign key to entity type B, and B has a foreign key to A. Circular references can occur with more extensive chains of foreign keys, such as A refers to B, which refers to C, which refers to A. The BillingCenter data model does not permit circular foreign keys reference, because BillingCenter cannot determine a safe order for committing the entity instances in a circular reference to the database.

For example, entity types A and B have foreign key references to each other. The foreign keys create a circular reference. Suppose that a bundle contains a new instance of A and a new instance of B. The circular reference would cause a foreign key constraint to fail upon committing the bundle. If BillingCenter commits A before B is committed and in the database, a constraint failure occurs on the foreign key from A to B. The converse order of committing B before A causes a similar failure.

An edge foreign key in place of a standard foreign key resolves circular references so BillingCenter can determine a safe order for committing the entity instances within a cycle. An edge foreign key from A to B introduces a new, hidden associative entity with a foreign key to A and a foreign key to B. The edge foreign key associates A and B without establishing foreign keys in the database directly between them. With an edge foreign key, BillingCenter can safely first commit new object A, then new object B, and finally the edge foreign key instance.

## Edge Foreign Keys in Entity Database Tables

Unlike a standard foreign key, an edge foreign key does not correspond to an actual column in the database table of an entity type. Nor does an edge foreign key implement a database foreign key constraint. However, the BillingCenter *Data Dictionary* labels edge foreign keys as standard foreign keys. In Gosu code, you access edge foreign keys in the same manner that you access standard foreign keys.

## Edge Foreign Keys and Associative Database Tables

An edge foreign key creates an *associative table* in the database. An associative table is essentially a table of foreign keys relationships. An associative table associates other database tables with each other but holds no other essential business data itself.

In BillingCenter, the associative table that implements an edge foreign key has two columns:

- OwnerID
- ForeignEntityID

If entity instance A has an edge foreign key to entity type B, BillingCenter creates a row in the edge foreign key table. The value in the row for OwnerID points to A and the value for ForeignEntityID points to B.

Every time you traverse, or dereference, the edge foreign key, BillingCenter 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 BillingCenter follows the ForeignEntityID on the row.

## Edge Foreign Keys in Gosu

In Gosu code, edge foreign keys work in a manner similar to standard foreign keys. Just like a standard foreign key, you can query an edge foreign key and get and set its attributes.

## Edge Foreign Keys and Performance

An edge foreign key has more performance issues than a standard foreign key, because BillingCenter must manage a separate table for the relationship. Queries must join an extra table. Nullability constraints in the database do not work with edge foreign keys, so you must enforce nullability constraints with extra Gosu code the you develop.

## Edge Foreign Keys and Archiving

If you add an edge foreign key to an entity that is part of the domain graph, the edge foreign key must also set its `extractable` attribute to `true`. Edge foreign keys do not inherit the `<implementsEntity>` delegate from their enclosing entities. If you do not mark edge foreign keys in extractable entities as themselves extractable, the server refuses to start.

### See also

## When to Use Edge Foreign Keys

Use an edge foreign key only to avoid circular foreign key references in the data model. Circular foreign key references can prevent BillingCenter from determining a safe order for committing the entity instances in a circular reference to the database.

Use an edge foreign key instead of standard foreign key in the following situations:

- An entity type has self-referencing foreign keys, including foreign keys between subtypes.
- A Group entity must specify its parent group.
- Entity type A has a foreign key to B, and entity type B has a foreign key to A.
- Cycles that involve more than two entity types.

- The primary member of an array requires a foreign key to its owner.

### Attributes of <edgeForeignKey>

The <edgeForeignKey> element contains the following attributes.

<edgeForeignKey> attribute	Description	Default
createhistogram	<p>Whether to create a histogram on the column during an update to the database statistics.</p> <p><b>Note:</b> It is possible to override this attribute on an existing column in an extension (*.etx) file using the &lt;column-override&gt; element. You can use the override to turn off an existing histogram or to create one that did not previously exist.</p> <p>This change does not take effect during an upgrade. The change occurs <i>only</i> if you regenerate statistics for the affected table by using the Guidewire maintenance_tools command.</p>	false
deprecated	<p>If true, then BillingCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.</p> <p>If you deprecate an item, use the description to explain why.</p> <p>For more information, see “The deprecated Attribute” on page 175.</p>	false
desc	A description of the purpose and use of the edge foreign key.	None
edgeTableName	The name of the edge table entity. If you do not specify one, then BillingCenter creates one automatically.	None
edgeTableName	<i>Required.</i> The name of the edge, or join array, table to create.	None
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
exportasid	If specified, BillingCenter exposes the field in SOAP APIs as a string, whose value represents the PublicID of the referenced object.	false
extractable	Whether the edge entity should be marked Extractable for archiving.	false
fkentity	<i>Required.</i> The entity to which this foreign key points.	None
getterScriptability	See “Data Objects and Scriptability” on page 156 for information.	all
ignoreforevents	<p>If you change (or add, or remove) an entity X that does not generate events, then BillingCenter searches for all event-generating entity instances that specify X. If BillingCenter 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, BillingCenter 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 BillingCenter ignores that link as it determines what entities specify another entity.</p> <ul style="list-style-type: none"> <li>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.</li> <li>At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events.</li> </ul>	false

<edgeForeignKey> attribute	Description	Default
<code>importableagainstexistingobject</code>	If true and the entity is importable, or loadable, then the value in the staging table can be a reference to an existing object. This reference is the publicID of a row in the source table for the referenced object.	true
<code>loadable</code>	If true, then BillingCenter creates a staging table for the edge table.	false
<code>loadedByCallback</code>	<i>Internal.</i> If true, then the loading code does not use a default value or report a warning if the column is nullable without a default.	false
<code>name</code>	<i>Required.</i> 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 edgeForeignKey objects.	true
<code>overlapTable</code>	Whether the edge entity should be marked OverlapTable for archiving.	false
<code>overwrittenInStagingTable</code>	<i>Internal.</i> If true and the edge table is loadable, the loader process auto-populates the staging table during import.  <b>IMPORTANT</b> If set to true, do not attempt to populate the table yourself, as the loader import process overwrites this table.	false
<code>setterScriptability</code>	See “Data Objects and Scriptability” on page 156 for information.	all
<code>soapnullok</code>	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	None

## Subelements of <edgeForeignKey>

**IMPORTANT** The <edgeForeignKey> element does not inherit the <implementsEntity> delegate from its enclosing entity. You must specify a value for the name attribute on <implementsEntity> if you wish to associate a delegate with this edge foreign key.

<edgeForeignKey> subelement	Attributes	Description
<code>fulldescription</code>	None	See “<fulldescription>” on page 188.
<code>tag</code>	None	See “<tag>” on page 194.

## <events>

If the <events> element appears within an entity, it indicates that the entity raises events. Usually, the code indicates the standard events (add, change, and remove) by default. If the <events> element does not appear in an entity, that entity does not raise any events. You cannot modify the set of the events associated with a base entity through extension. However, you can add additional events to a base entity through extension, even if that entity already contains a set of predefined events.

**Note:** This element is not valid for a nonPersistentEntity.

Guidewire defines this element in the data model metadata files as the <events> XML subelement. There can be at most one <events> element in an entity. However, you can specify additional events through the use of <event> subelements. For example:

```
<events>
  <event>
    ...
  </events>
```

**Note:** BillingCenter automatically adds the EventAware delegate to any entity that contains the <events> element.

### Attributes of <events>

There are no attributes on the <events> element.

### Subelements of <events>

The <events> element contains the following subelements.

<events> subelement	Description
event	<p>Defines an additional event to fire for the entity. Use multiple &lt;event&gt; elements to specify multiple events. This subelement contains the following attributes:</p> <ul style="list-style-type: none"> <li>• description (required = true)</li> <li>• name (required = true)</li> </ul> <p>The attributes are self-explanatory. The &lt;event&gt; element requires each one.</p>

## <foreignkey>

The <foreignkey> element defines a foreign key reference to another entity.

### Attributes of <foreignkey>

The <foreignkey> element contains the following attributes.

<foreignkey> attribute	Description	Default
columnName	<p><i>Optional.</i> If specified, BillingCenter uses this value as the column name of the corresponding database column. If you do not specify a columnName value, then BillingCenter uses the value of the name attribute for the database column name.</p> <p><b>Note:</b> As a common and recommended practice, use the suffix ID for the column name. For example, for a foreign key with name Account, set the columnName to AccountID.</p> <p>Guidewire does not require that you use an ID suffix on names of foreign key columns. However, Guidewire strongly recommends that you adopt this practice to help you analyze the database and identify foreign keys.</p> <p><b>IMPORTANT</b> All column names on a table must be unique in that table. Otherwise, Studio displays an error if you verify the resource, and the application server fails to start.</p>	None
createConstraint	If true, the database creates a foreign key constraint for this foreign key.	true
createBackingIndex	If true, the database automatically creates a backing index on the foreign key. If set to false, the database does not create a backing index.	true
createHistogram	<p>Whether to create a histogram on the column during an update to the database statistics.</p> <p><b>Note:</b> It is possible to override this attribute on an existing column in an extension (*.etx) file using the &lt;columnOverride&gt; 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 only if you regenerate statistics for the affected table by using the Guidewire maintenance_tools command.</p>	false
<p><b>See also</b></p> <ul style="list-style-type: none"> <li>• “Working with Attribute Overrides” on page 210</li> <li>• “Configuring Database Statistics” on page 39 in the <i>System Administration Guide</i></li> <li>• “Maintenance Tools Command” on page 188 in the <i>System Administration Guide</i></li> </ul>		

<foreignkey> attribute	Description	Default
deprecated	If true, then BillingCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.  If you deprecate an item, use the description to explain why.  For more information, see "The deprecated Attribute" on page 175.	false
desc	A description of the purpose and use of the field.	None
existingreferencesallowed	If the following attributes are set to false, which is not the default: <ul style="list-style-type: none"><li>• loadable</li><li>• importableagainstexistingobject</li></ul> then, the value in the staging table can only be a reference to an existing object.	true
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
exportasid	If specified, BillingCenter exposes the field in SOAP APIs as a string, whose value represents the PublicID of the referenced object.	false
fkentity	<i>Required.</i> The entity to which this foreign key refers.	None
getterScriptability	See "Data Objects and Scriptability" on page 156 for information.	all
ignoreforevents	If you change (or add, or remove) an entity X that does not generate events, then BillingCenter searches for all event-generating entity instances that specify X. If BillingCenter 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, BillingCenter 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 BillingCenter ignores that link as it determines what entities specify another entity. <ul style="list-style-type: none"><li>• 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.</li><li>• At the column level, the ignoreForEvents attribute means changes to this column do not cause the application to generate events.</li></ul>	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
LoadedByCallback	<i>Internal.</i> If true, then the loading code does not use a default value or report a warning if the column is nullable without a default.	false
name	<i>Required.</i> Specifies the name of the property on the entity.	None
nullok	Whether the field can contain null values.	true

<foreignkey> attribute	Description	Default
overwrittenInStagingTable	<i>Internal.</i> If true (and the table is loadable), it indicates that the loader process auto-populates the staging table during import.  <b>IMPORTANT</b> If set to true, do not attempt to populate the table yourself because the loader import process overwrites this table.	false
owner	If true, it indicates that even if it is a foreign key, the row from the other table that this key references is a child node.	false
setterScriptability	See "Data Objects and Scriptability" on page 156 for information.	all
soapnullok	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	None
triggersValidation	Whether changes to the entity referred to by this foreign key trigger validation.	false

### Subelements of <foreignkey>

The <foreignkey> element contains the following subelements.

<foreignkey> subelement	Attributes	Description
fulldescription	None	See "<fulldescription>" on page 188.
tag	None	See "<tag>" on page 194.

## <fulldescription>

BillingCenter uses the fulldescription subelement to populate the *Data Dictionary*. For example:

```
<fulldescription>
  <![CDATA[<p>Aggregates the information needed to display one activity row
  (base entity for all other activity views).</p>]]>
</fulldescription>
```

## <implementsEntity>

The <implementsEntity> subelement specifies that an entity implements the specified delegate. Guidewire calls an entity an *implementor* of a delegate if the entity specifies the delegate in a <implementsEntity> subelement.

**IMPORTANT** Do not change the delegate that a Guidewire base entity implements by creating an extension entity that includes an <implementsEntity> subelement. BillingCenter generates an error if you do.

If a delegate definition includes the optional requires attribute, then the implementor must provide an adapter attribute on its <implementsEntity> subelement. The adapter attribute specifies the name of a Java or Gosu type that implements the interface that the delegate definition specifies in its own requires attribute.

For example, the PolicyCenter base configuration defines a Cost delegate as follows:

```
<?xml version="1.0"?>
<delegate ... name="Cost" requires="gw.api.domain.financials.CostAdapter">
  ...
</delegate>
```

The base configuration defines a BACost entity that includes an <implementsEntity> subelement, which specifies delegate with name="Cost". Therefore, the BACost entity is an implementor of the Cost delegate.

```
<?xml version="1.0"?>
<entity ... entity="BACost" ... >
  ...
  <implementsEntity name="Cost" adapter="gw.lob.ba.financials.BACostAdapter" />
  ...
</entity>
```

The `Cost` delegate requires an implementation of the `CostAdapter` interface. So in its `adapter` attribute, the `BACost` entity specifies a `BACostAdapter` class, which implements the `CostAdapter` interface that the `Cost` adapter specifies in its `requires` attribute.

Follow these rules for defining entities that implement delegates:

- If you specify a value for the `requires` attribute in a delegate, then implementers of the delegate must specify an `adapter` attribute in their definitions. The `adapter` attribute must specify the name of a Java or Gosu type that implements the interface specified by the `requires` attribute in delegate definition.
- If you do not specify a value for the `requires` attribute in a delegate, then implementers of the delegate must not specify an `adapter` attribute their definitions.

### The Extractable Delegate

Entities that are part of the domain graph must implement the `Extractable` delegate. If you add an edge foreign key to an entity that is part of the domain graph, then the edge foreign key must set the `extractable` attribute to `true`.

For example, if you create a custom subtype of `Contact`, then the custom subtype must implement the `Extractable` delegate. Edge foreign keys do not inherit delegate definitions from their enclosing entity.

---

**WARNING** Entities that are part of the domain graph must implement the `Extractable` delegate by using the `<implementsEntity>` element. Otherwise, the server refuses to start.

---

### Attributes of `<implementsEntity>`

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

<code>&lt;implementsEntity&gt;</code> subelement	Description
<code>adapter</code>	The name of the type that implements the interface specified by the <code>requires</code> attribute on <code>&lt;delegate&gt;</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 of `<implementsEntity>`

There are no subelements on the `<implementsEntity>` subelement.

## `<implementsInterface>`

The `<implementsInterface>` subelement specifies that an entity implements the specified interface. This element defines two attributes, an `interface` (`iFace`) attribute and an `implementation` (`impl`) attribute. The `<implementsInterface>` subelement requires both attributes.

For example, the PolicyCenter base configuration defines the `BACost` entity with the following `<implementsInterface>` subelement:

```
<entity ... entity="BACost" ...
  ...
  <implementsInterface
    iFace="gw.lob.ba.financials.BACostMethods"
    impl="gw.lob.ba.financials.BACostMethodsImpl"/>
</entity>
```

The `BACostMethods` interface has getter methods that any class which implements this interface must provide. The getter methods are `coverage`, `state`, and `vehicle`. By including the `<implementsInterface>` subelement, the `BACost` entity lets you use getter methods on instances of the `BACost` entity in Gosu code.

```
var cost : BACost
var cov      = cost.Coverage
```

```
var state = cost.State
var vehicle = cost.Vehicle
```

#### Attributes of <implementsInterface>

The <implementsInterface> element contains the following attributes.

<implementsInterface> subelement	Description
iface	<i>Required.</i> The name of the interface that this data object must implement.
impl	<i>Required.</i> The name of the class or subclass that implements the specified interface.

#### Subelements on <implementsInterface>

There are no subelements on the <implementsInterface> subelement.

## <index>

The <index> element defines an index on the database table used to store the data for an entity. Guidewire defines this element in the data model metadata files as the <index> XML subelement. This element contains a required subelement, which is <indexcol>.

The <index> element instructs BillingCenter to create an index on the physical database table. This index is in addition to those indexes that BillingCenter 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, BillingCenter frequently searches non-retired trouble tickets for one with a particular number. Therefore, the `TroubleTicket` entity defines an index containing both the `Retired` and `TroubleTicketNumber` fields. However, another common search uses just `TroubleTicketNumber`. Since that field is already part of another index, a separate index containing only `TroubleTicketNumber` is unnecessary.

A column used in an index cannot have a length of more than 1000 characters.

---

**IMPORTANT** In general, the use of a database index has the possibility of reducing update performance. Guidewire recommends that you add a database index with caution. In particular, do not attempt to add an index on a column of type CLOB or BLOB. If you do so, BillingCenter generates an error message upon resource verification.

---

#### Attributes of <index>

The <index> element contains the following attributes.

<index> attribute	Description	Default
desc	A description of the purpose and use of the index.	None
expectedtobecovering	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 desc attribute to indicate which operation that is.	false

<index> attribute	Description	Default
name	<p><i>Required.</i> The name of the index. The first character of the name must be a letter. The maximum length for an index name is 18 characters.</p> <p><b>IMPORTANT</b> For &lt;subtype&gt; definitions, all index names must be unique between the subtype and supertype. In other words, do not duplicate an index name between the subtype definition in the extensions folder and its supertype in the metadata folder. Otherwise, BillingCenter generates an error on resource verification.</p>	None
trackUsage	If true, track the usage of this index.	true
unique	Whether the values of the index are unique for each row.	false
verifyInLoader	If true, then BillingCenter runs an integrity check for unique indexes before loading data from the staging tables.	true

### Subelements of <index>

The <index> element contains the following subelements.

<index> subelement	Description	Default
forceindex	<p>Use to force BillingCenter to create an index if running against a particular database.</p> <p>This subelement is useful because the index generation algorithm can throw away some declared indexes as being redundant. In some cases, BillingCenter 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"> <li>• oracle – If true, force the creation of an index if running against an Oracle database.</li> <li>• sqlserver – If true, force the creation of an index if running against a Microsoft SQL Server database.</li> </ul>	None
indexcol	<p><i>Required.</i> Defines a field that is part of the index. You can specify multiple &lt;indexcol&gt; elements to define composite indexes. This subelement contains the following attributes:</p> <ul style="list-style-type: none"> <li>• keyposition – <i>Required.</i> The position of the field within the index. The first position is 1.</li> <li>• name – <i>Required.</i> The column name of the field. This name can be a column, foreignkey, or typekey defined in the entity.</li> <li>• sortascending – If true, the default, then the sort direction is ascending. If false, then the sort direction is descending.</li> </ul> <p>The column cannot have a length of more than 1000 characters.</p>	None

### <onetoone>

The <onetoone> element defines a single-valued association to another entity that has a one-to-one cardinality. Guidewire defines this element in the data model metadata files as the <onetoone> XML subelement. A one-to-one element functions in a similar manner to a foreign key in that it makes a reference to another entity. However, its purpose is to provide a reverse pointer to an entity or object that is pointing at the <onetoone> entity, through the use of a foreign key.

For example, entity A has a foreign key to entity B. You can associate an instance of B with at most one instance of A. Perhaps, there is a unique index on the foreign key column. This then defines a one-to-one relationship between A and B. You can then declare the `<onetoone>` element on B, to provide simple access to the associated A. In essence, using a one-to-one element creates an *array-of-one*, with, at most, one element. Zero elements are also possible.

**Note:** BillingCenter labels one-to-one elements in the Guidewire *Data Dictionary* as foreign keys. You access these elements in Gosu code in the same manner as you access foreign keys.

#### Attributes of `<onetoone>`

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

<code>&lt;onetoone&gt;</code> attribute	Description	Default
<code>cascadeDelete</code>	If true, then BillingCenter deletes the entity to which the <code>&lt;onetoone&gt;</code> element points if you delete this entity.	false
<code>deprecated</code>	If true, then BillingCenter marks the item as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.  If you deprecate an item, use the description to explain why.  For more information, see “The deprecated Attribute” on page 175.	false
<code>desc</code>	A description of the purpose and use of the field.	None
<code>exportable</code>	<i>Ddeprecated</i> . Only used with RPCE web services, which are deprecated.	true
<code>fkentity</code>	<i>Required</i> . The entity to which this foreign key points.	None
<code>getterScriptability</code>	See “Data Objects and Scriptability” on page 156 for information.	all
<code>ignoreforevents</code>	If you change (or add, or remove) an entity X that does not generate events, then BillingCenter searches for all event-generating entity instances that specify X. If BillingCenter 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, BillingCenter 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 BillingCenter ignores that link as it determines what entities specify another entity. <ul style="list-style-type: none"> <li>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.</li> <li>At the column level, the <code>ignoreForEvents</code> attribute means changes to this column do not cause the application to generate events.</li> </ul>	false
<code>linkField</code>	<i>Optional</i> . Specifies the foreign key field that points back to this object.	None
<code>name</code>	<i>Required</i> . Specifies the name property on the entity.	None
<code>nullok</code>	Whether the field can contain null values.	true
<code>owner</code>	If true, this entity owns the linked object (the object to which the <code>&lt;onetoone&gt;</code> element points): <ul style="list-style-type: none"> <li>If you delete the owning object, then BillingCenter deletes the linked object as well.</li> <li>If you update the object pointed to by the <code>&lt;onetoone&gt;</code> element, then BillingCenter considers the owning object updated as well.</li> </ul>	false
<code>setterScriptability</code>	See “Data Objects and Scriptability” on page 156 for information.	all
<code>triggersValidation</code>	Whether changes to the entity pointed to by this entity trigger validation.	false

### Subelements of <onetoone>

The <onetoone> element contains the following subelements.

<onetoone> subelement	Description	Default
fulldescription	See “<fulldescription>” on page 188.	None
tag	See “<tag>” on page 194.	None

## <remove-index>

The <remove-index> element defines the name of a database index that you want to remove from the data model. It is valid for use with the following data model elements:

- <entity>
- <extension>

You can use this element to safely remove a non-primary key index if it is one of the following:

- non-unique
- unique but contains an ID column

Guidewire performs metadata validation to ensure that the <remove-index> element removes only those indexes that fall into one of these categories.

### The Index is Non-unique

You can safely remove a non-primary key index with the `unique` attribute set to `false`. In general, these are indexes that Guidewire provides for performance enhancement. It is safe to remove these kinds of indexes.

### The Index is Unique, But Contains an ID Column

You can safely remove a non-primary key index with the `unique` attribute set to `true` if that index includes ID as a key column. For example, the `WorkItem` entity contains the following index definition:

```
<index desc="Covering index to speed up checking-out of work items and they involve search on status"
       name="WorkItemIndex2" unique="true">
  <indexcol keyposition="1" name="status"/>
  <indexcol keyposition="2" name="Priority" sortascending="false"/>
  <indexcol keyposition="3" name="CreationTime"/>
  <indexcol keyposition="4" name="ID"/>
</index>
```

Even though the `unique` attribute is set to `true`, you can safely remove this index because the index definition contains an ID column, `keyposition="4"`. These types of indexes do not enforce a uniqueness condition. Thus, it is safe to remove these kinds of indexes.

### Attributes of <remove-index>

The <remove-index> element contains the following attributes.

<remove-index> attribute	Description	Default
name	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.

## <tag>

The <tag> element defines a data model annotation. This allows you to define your own metadata to add to the data model.

The file `BillingCenter/modules/configuration/config/metadata/tags.lst` specifies the list of valid tags that you can use. To add a new tag, edit this file, and then add the tag name on a new line.

At most one of each tag can exist on a field at one time. Extensions can add, but not override, tags on existing fields.

To retrieve this data at run time, use the `DatamodelTags` property on `IEntityPropertyInfo`, the metadata object for entity properties. For example:

```
// On the entity info metadata, get the metadata for the properties on this entity.  
// This type is an iterator of objects of type IEntityPropertyInfo.  
var props = User.Type.EntityProperties  
  
// Get the property info object by its name.  
var theProp = props.toList().firstWhere( \ PropertyInfo -> PropertyInfo.ColumnName == "Language")  
  
// Get the data model tags, which has the type java.util.Map  
var myTags = theProp.DatamodelTags  
  
// Get a specific tag  
var theTag = myTags["NameOfTag"]
```

### Attributes of <tag>

The <tag> element contains the following attributes.

<tag> attribute	Description	Default
name	<i>Required.</i> The name of the tag.	None
value	The value of the tag.	None

## <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 245.

### Attributes of <typekey>

The <typekey> element contains the following attributes.

<typekey> attribute	Description	Default
columnName	<p><i>Optional.</i> If specified, BillingCenter uses this value as the column name of the corresponding database column. If you do not specify a columnName value, then BillingCenter uses the value of the name attribute for the database column name.</p> <p><b>IMPORTANT</b> All column names on a table must be unique within that table. Otherwise, Studio displays an error if you verify the resource and the application server fails to start.</p>	None
createhistogram	<p>Whether to create a histogram on the column during an update to the database statistics.</p> <p><b>Note:</b> It is possible to override this attribute on an existing column in an extension (*.etx) file using the &lt;column-override&gt; 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><b>See also</b></p> <ul style="list-style-type: none"> <li>• “Working with Attribute Overrides” on page 210</li> <li>• “Configuring Database Statistics” on page 39 in the <i>System Administration Guide</i></li> <li>• “Maintenance Tools Command” on page 188 in the <i>System Administration Guide</i></li> </ul>	false
default	The default value given to the field during new entity creation.	None
deprecated	<p>If true, then BillingCenter marks the typekey as deprecated in the <i>Data Dictionary</i> and places a Deprecated annotation on it in the Guidewire Studio API Reference.</p> <p>If you deprecate a typekey, use the description attribute (desc) to explain why.</p> <p>For more information, see “The deprecated Attribute” on page 175.</p>	false
desc	A description of the purpose and use of the field.	None
exportable	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	true
getterScriptability	See “Data Objects and Scriptability” on page 156 for information.	None
loadable	If true, then you can load the field through staging tables. A staging table can contain a column, as a String, for the code of the typekey.	true
loadedByCallback	<i>Internal.</i> If true, then the loading code does not use a default value or report a warning if the column is nullable without a default.	false
name	<i>Required.</i> Specifies the name of the property on the entity	None
nullok	Whether the field can contain null values.	true
overwrittenInStagingTable	<i>Internal.</i> If true and the typekey is loadable, the loader process auto-populates the typekey in the staging table during import.	false
	<b>IMPORTANT</b> If set to true, do not attempt to populate the typekey yourself because the loader import process overwrites this typekey.	
setterScriptability	See “Data Objects and Scriptability” on page 156 for information.	None
soapnullok	<i>Deprecated.</i> Only used with RPCE web services, which are deprecated.	None
typefilter	The name of a filter associated with the typelist. See “Static Filters” on page 256 for additional information.	None
typelist	<i>Required.</i> The name of the typelist from which this field gets its value.	None
	<b>See also</b>	
	“Working with Typelists” on page 245.	

### Subelements of <typekey>

The <typekey> element contains the following subelements.

<typekey> subelement	Description	Default
keyfilters	Defines one or more <keyfilter> elements. There can be at most one <keyfilters> element in an entity. See “Dynamic Filters” on page 260 for additional information.	None
fulldescription	See “<fulldescription>” on page 188.	None
tag	See “<tag>” on page 194.	None

### Subelements of <keyfilters>

The <keyfilters> element contains the following subelements.

<keyfilters> subelement	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.  <b>Note:</b> You can include multiple <keyfilter> elements to specify multiple keyfilters.	None

# Working with Associative Arrays

This topic describes the different types of associative arrays that Guidewire provides as part of the base data model configuration.

This topic includes:

- “Overview of Associative Arrays” on page 197
- “Subtype Mapping Associative Arrays” on page 199
- “Typelist Mapping Associative Arrays” on page 201

## 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  
...
```

BillingCenter uses associate arrays to expose array values as a typesafe map within Gosu code. The following example uses a typekey from a State typelist as the mapping index for an associative array of state capitals:

State typekey index	Maps to...
Capital[State.TC_AL]	Montgomery
Capital[State.TC_AK]	Juneau
Capital[State.TC_AZ]	Phoenix
Capital[State.TC_AR]	Little Rock

There are two necessary tasks in working with an associative array in Gosu:

- Exposing the key set to the type system
- Calculating the value from the key

## Associative Array Mapping Types

An associative array must have a key that maps to a value. The mapping type describes what BillingCenter uses as the key and what value that key returns.

Mapping type	Key	Value
Subtype mapping	Entity subtype	Implicit subtype field on an entity
TypeList mapping	TypeList	Typekey field on the entity

To implement an associative array, add one of the following elements to an `<array>` element in the data type definition file. The number of results that each returns—the cardinality of the result set—depends on the element type.

<code>&lt;link-association&gt;</code>	Returns at most one element. The return type is an object of the type of the array.
<code>&lt;array-association&gt;</code>	Returns an array of results that match the typekey. The number of results can be zero, one, or more.

Each `<array>` element in a data type definition file can have zero to one of each of these elements.

As an example, in the ClaimCenter `Claim` definition file (`configuration → config → Metadata → Entity → Claim.eti`), you see the following XML (simplified for clarity):

```

<entity xmlns="http://guidewire.com/datamodel"
        entity="Claim"
        table="claim"
        type="retireable">
    ...
    <array arrayentity="ClaimMetric"
           desc="Metrics related to this claim."
           exportable="false"
           ignoreforevents="true"
           name="ClaimMetrics"
           triggersValidation="false">
        <link-association>
            <subtype-map/>
        </link-association>
        <array-association>
            <typeList-map field="ClaimMetricCategory"/>
        </array-association>
    </array>
    ...
</entity>

```

### See also

- For examples of how to create a subtype associative array, see “Subtype Mapping Associative Arrays” on page 199.
- For examples of how to create a typeList associative array, see “TypeList Mapping Associative Arrays” on page 201.

## Scriptability and Associative Arrays

It is possible to set the following attributes on each `<link-association>` and `<array-association>` element:

- `hasGetter`
- `hasSetter`

For example:

```
<link-association hasGetter="true" hasSetter="true">
  <typelist-map field="TAccountType"/>
</link-association>
```

For these attributes:

- If `hasGetter` is `true`, then you can read the property.
- If `hasSetter` is `true`, then you can update the property.

**Note:** If you do not specify either of these attributes, then BillingCenter defaults to `hasGetter="true"`.

#### See also

- “Data Objects and Scriptability” on page 156

## Issues with Setting Array Member Values

There are several issues with setting associative array member values, including:

1. You can use a query builder expression to retrieve a specific entity instance. However, the result of the query is read-only. You must add the retrieved entity to a bundle to be able to manipulate its fields. To work with bundles, use one of the following:

```
var bundle = gw.transaction.Transaction.getCurrent()
gw.transaction.Transaction.runWithNewBundle(\ bundle -> ) //Use this version in the Gosu tester
```

2. You can only set array values on fields that are database-backed fields, not fields that are derived properties. To determine which fields are derived, consult the BillingCenter *Data Dictionary*.

#### See also

- See “Overview of the Query Builder APIs” on page 129 in the *Gosu Reference Guide* for information on working with query builder expressions.

## Subtype Mapping Associative Arrays

You use subtype mapping to access array elements based on their subtype. In other words, this type of associative array divides the elements of the array into multiple partitions, each of which contains only array elements of a particular object *subtype*. For example, in the ClaimCenter base configuration, the data model defines an associative array called `ClaimMetrics` on the `Claim` object.

In the `Claim` definition file (`configuration → config → Metadata → Entity → Claim.eti`), you see the following (simplified) XML:

```
<entity xmlns="http://guidewire.com/datamodel"
  entity="Claim"
  table="claim"
  type="retireable">
  ...
  <array arrayentity="ClaimMetric"
    desc="Metrics related to this claim."
    exportable="false"
    ignoreforevents="true"
    name="ClaimMetrics"
    triggersValidation="false">
    <link-association>
      <subtype-map/>
    </link-association>
  </array>
  ...
</entity>
```

The array—`ClaimMetrics`—contains a number of objects, each of which is a subtype of a `ClaimMetric` object. The data model defines the associative array using the `<link-association>` element. A link associations return

at most one element and the return type is an object of the type of the array. In this case, the return type is an object of type `ClaimMetric`, or more specifically, one of its subtypes.

The ClaimCenter data model defines a number of subtypes of the `ClaimMetric` object, including:

- `DecimalClaimMetric`
- `IntegerClaimMetric`
- `AllEscalatedActivitiesClaimMetric`
- `OpenEscalatedActivitiesClaimMetric`
- ...

To determine the complete list of subtypes on an object, consult the *Data Dictionary*. The dictionary organizes the subtypes into a table at the top of the dictionary page with active links to sections that describe each subtype in greater detail.

## Working with Array Values Using Subtype Mapping

To retrieve an array value through subtype mapping, use the following syntax:

```
base-entity.subtype-map.property
```

Each field has the following meanings:

<code>base-entity</code>	The base object on which the associative array exists, for example, the <code>Claim</code> entity for the <code>ClaimMetrics</code> array.
<code>subtype-map</code>	The array entity subtype, for example, <code>AllEscalatedActivitiesClaimMetric</code> (a subtype of <code>ClaimMetric</code> ).
<code>property</code>	A field or property on the array object. For example, the <code>AllEscalatedActivitiesClaimMetric</code> object contains the following properties (among others): <ul style="list-style-type: none"> <li>• <code>ClaimMetricCategory</code></li> <li>• <code>DisplayTargetValue</code></li> <li>• <code>DisplayValue</code></li> </ul>

**Note:** To see a list of subtypes for any given object, consult the BillingCenter *Data Dictionary*. To determine the list of fields (properties) on an object, again consult the *Data Dictionary*.

### Example One

The following example code uses the sample data in the Guidewire ClaimCenter base configuration. It first retrieves a specific claim object using a query builder and then uses that object as the base entity from which to retrieve array member properties.

```
var clm = gw.api.database.Query.make(Claim).compare("ClaimNumber", Equals,
  "235-53-365870").select().getAtMostOneRow()

print("AllEscalatedActivitiesClaimMetric\tClaim Metric Category = "
  + clm.AllEscalatedActivitiesClaimMetric.ClaimMetricCategory.DisplayName)
print("AllEscalatedActivitiesClaimMetric\tDisplay Value = "
  + clm.AllEscalatedActivitiesClaimMetric.DisplayValue)
print("AllEscalatedActivitiesClaimMetric\tReach Yellow Time = "
  + clm.AllEscalatedActivitiesClaimMetric.ReachYellowTime)
```

The output of running this code in the Gosu tester looks similar to the following:

```
AllEscalatedActivitiesClaimMetric      Claim Metric Category = Claim Activity
AllEscalatedActivitiesClaimMetric      Display Value = 0
AllEscalatedActivitiesClaimMetric      Reach Yellow Time = null
```

### Example Two

The following sample code:

- Retrieves a read-only claim object.
- Adds the claim object to transaction bundle to make it writable.

- Sets a specific property on the `AllEscalatedActivitiesClaimMetric` object (a subtype of the `ClaimMetric` object) associated with the claim.

If you recall from the definition of the `claim` object, `ClaimCenter` associates an array of `ClaimMetric` objects—the `ClaimMetrics` array—with the `Claim` object. The metadata definition file also defines the `ClaimMetrics` array as being of type `<link-association>` using subtypes. Thus, you can access array member properties by first accessing the array member of the proper subtype.

```
uses gw.transaction.Transaction

var todaysDate = java.util.Date.getCurrentDate
var clm = gw.api.database.Query.make(Claim).compare("ClaimNumber", Equals,
    "235-53-365870").select().getAtMostOneRow()

//Query result is read-only, need to get current bundle and add object to bundle
var bundle = Transaction.getCurrent()
clm = bundle.add(clm)

print("AllEscalatedActivitiesClaimMetric\tReach Yellow Time = "
    + clm.AllEscalatedActivitiesClaimMetric.ReachYellowTime)
clm.AllEscalatedActivitiesClaimMetric.ReachYellowTime = todaysDate

print("\nAfter modifying the ReachYellowTime value...\n")
print("AllEscalatedActivitiesClaimMetric\tReach Yellow Time = "
    + clm.AllEscalatedActivitiesClaimMetric.ReachYellowTime)
```

The output of running this code in the `Gosu` tester looks similar to the following:

```
AllEscalatedActivitiesClaimMetricReach Yellow Time = null

After modifying the ReachYellowTime value...

AllEscalatedActivitiesClaimMetricReach Yellow Time = 2010-05-21
```

For more information making query results writable, see “Adding Entity Instances to Bundles” on page 345 in the *Gosu Reference Guide*.

## TypeList Mapping Associative Arrays

You use a typelist map to partition array objects based on a typelist field (typecode) in the `<array>` element. In the `ClaimCenter` base configuration, the `ClaimMetrics` array on `Claim` contains a typelist mapping and the previously described subtype mapping.

```
<entity xmlns="http://guidewire.com/datamodel"
    entity="Claim"
    table="claim"
    type="retireable">
    ...
    <array arrayentity="ClaimMetric"
        desc="Metrics related to this claim."
        exportable="false"
        ignoreforevents="true"
        name="ClaimMetrics"
        triggersValidation="false">
        <array-association>
            ...
            <typelist-map field="ClaimMetricCategory"/>
        </array-association>
    </array>
    ...
</entity>
```

The `<typelist-map>` element requires that you set a value for the `field` attribute. This attribute specifies the typelist to use to partition the array.

**IMPORTANT** It is an error to specify a typelist mapping on a field that is not a typekey.

Associative arrays of type <array-association> are different from those created using <link-association> in that they can return more than a single element. In this case, the code creates an array of `ClaimMetric` objects named `ClaimMetrics`. Each `ClaimMetric` object, and all subtype objects of it, contain a property called `ClaimMetricCategory`. The array definition code utilizes that fact and uses the `ClaimMetricCategory` typelist as a partitioning agent.

The `ClaimMetricCategory` typelist contains three typecodes, which are:

- `ClaimActivityMetrics`
- `ClaimFinancialMetrics`
- `OverallClaimMetrics`

Each typecode specifies a category, which contains multiple `ClaimMetric` object subtypes. For example, the `OverallClaimMetrics` category contains two `ClaimMetric` subtypes:

- `DaysInitialContactWithInsuredClaimMetric`
- `DaysOpenClaimMetric`

In another example from the ClaimCenter base configuration, you see the following defined for `ReserveLine`.

```
<entity entity="ReserveLine"
  xmlns="http://guidewire.com/datamodel"
  ...
  table="reserveline"
  type="retireable">
  ...
  <array arrayentity="TAccount"
    arrayfield="ReserveLine"
    name="TAccounts"
    ...
    <link-association hasGetter="true" hasSetter="true">
      <typelist-map field="TAccountType"/>
    </link-association>
  </array>
  ...
</entity>
```

In this case, the array definition code creates a <link-association> array of `TAccount` objects and partitions the array by the `TAccountType` typelist typecodes.

## Working with Array Values Using Typelist Mapping

To retrieve an array value through typelist mapping, use the following syntax:

```
entity.typecode.property
```

Each field has the following meaning:

<code>entity</code>	The object on which the associative array exists, for example, the <code>ReserveLine</code> entity on which the <code>Taccounts</code> array exists
<code>typecode</code>	The typelist typecode that delimits this array partition, for example, <code>OverallClaimMetrics</code> (a typecode from the <code>ClaimMetricCategory</code> typelist).
<code>property</code>	A field or property on the array object. For example, the <code>ClaimMetric</code> object contains the following properties (among others): f <ul style="list-style-type: none"> <li>• <code>ReachRedTime</code></li> <li>• <code>ReachYellowTime</code></li> <li>• <code>Skipped</code></li> </ul>

### Example One

The following example code uses the sample data in the Guidewire ClaimCenter base configuration. It iterates over the members of the `ClaimMetrics` array that fall into the `OverallClaimMetrics` category. (The `ClaimMetricCategory` typelist contains multiple type codes, of which `OverallClaimMetrics` is one.)

```
uses gw.api.database.Query

var clm = Query.make(Claim).compare("ClaimNumber", Equals, "235-53-365870").select().FirstResult
for (time in clm.OverallClaimMetrics) {
```

```

        print(time.Subtype.DisplayName + ": ReachYellowTime = " + time.ReachYellowTime)
    }

```

The output of running this code in the Gosu tester looks something similar to the following:

```

Initial Contact with Insured (Days): ReachYellowTime = 2010-09-27
Days Open: ReachYellowTime = 2011-04-08

```

### Example Two

The following example code also uses the sample data in the Guidewire ClaimCenter base configuration. It first retrieves a specific `Claim` object and then retrieves a specific `ReserveLine` object associated with that claim.

```

var clm = gw.api.database.Query.make(Claim).compare("ClaimNumber", Equals,
    "235-53-365870").select().FirstResult
var thisReserveLine = clm.ReserveLines.first()

print(thisReserveLine)
print(thisReserveLine.cashout.CreateTime)

```

The output of running this code in the Gosu tester looks something similar to the following:

```

(1) 1st Party Vehicle - Ray Newton; Claim Cost/Auto body
Fri Oct 08 16:14:50 PDT 2010

```

### Setting Array Member Values

The following example code also uses the sample data in the Guidewire ClaimCenter base configuration. It uses a query builder expression to retrieve a specific claim entity. As the result of the query is read-only, you must first retrieve the current bundle, then add the claim to the bundle to make its fields writable. The retrieved claim is the base entity on which the `ClaimMetrics` array exists.

The following sample code:

- Retrieves a read-only claim object.
- Adds the claim object to transaction bundle to make it writable.
- Sets specific properties on the `ClaimMetric` object associated with the claims that are in the `OverallClaimMetrics` category.

If you recall from the definition of the claim object, ClaimCenter associates an array of `ClaimMetric` objects—the `ClaimMetrics` array—with the `Claim` object. The metadata definition file also defines the `ClaimMetrics` array as being of type `<array-association>` using the `ClaimMetricCategory` typelist. Thus, you can access array member properties by first accessing the array member of the proper category.

```

uses gw.transaction.Transaction
uses gw.api.database.Query

var todaysDate = java.util.Date.getCurrentDate
var thisClaim = Query.make(Claim).compare("ClaimNumber", Equals, "235-53-365870").select().FirstResult
//Query result is read-only, need to get current bundle and add entity to bundle

var bundle = Transaction.getCurrent()
thisClaim = bundle.add(thisClaim)

//Print out the current values for the ClaimMetric.ReachYellowTime field on each subtype
for (color in thisClaim.OverallClaimMetrics) {
    print("Subtype - " + color.Subtype.DisplayName + ": ReachYellowColor = " + color.ReachYellowTime)
}

print("\nAfter modifying the values...\n")

//Modify the ClaimMetric.ReachYellowColor value and print out the new values
for (color in thisClaim.OverallClaimMetrics) {
    color.ReachYellowTime = todaysDate
    print("Subtype - " + color.Subtype.DisplayName + ": ReachYellowColor = " + color.ReachYellowTime)
}

```

The output of running this code in the Gosu tester looks similar to the following:

```

Subtype - Initial Contact with Insured (Days): ReachYellowColor = 2010-10-13
Subtype - Days Open: ReachYellowColor = 2010-10-13

After modifying the values...

```

```
Subtype - Initial Contact with Insured (Days): ReachYellowColor = 2010-10-13
Subtype - Days Open: ReachYellowColor = 2010-10-13
```

For more information making query results writable, see “Adding Entity Instances to Bundles” on page 345 in the *Gosu Reference Guide*.

# Modifying the Base Data Model

This topic discusses how to extend the base data model as well as how to create new data objects.

This topic includes:

- “Planning Changes to the Base Data Model” on page 205
- “Defining a New Data Entity” on page 208
- “Extending a Base Configuration Entity” on page 209
- “Working with Attribute Overrides” on page 210
- “Extending the Base Data Model: Examples” on page 212
- “Removing Objects from the Base Configuration Data Model” on page 219
- “Deploying Data Model Changes to the Application Server” on page 223

## Planning Changes to the Base Data Model

Before proceeding to modify the base data model, Guidewire strongly recommends that you first review the *Data Dictionary*. Verify that the existing data model does not provide the functionality that you need first before modifying the base application functionality.

### Overview of Data Model Extension

Entity extensions are additions to the entities in the base data model. Although you cannot modify the base data type declaration files directly, you can define an extension to one in a separate .etx file. You can also define new data model objects that extend the data model in an .eti file. This allows new BillingCenter releases to modify the base definitions without affecting your extensions, thus preserving an upgrade path.

By extending the base data model, you can:

- Add fields (columns) to an existing base entity through the use of the <column>, <typekey>, <foreignkey>, <array>, and similar elements. See “Data Object Subelements” on page 174.
- Create a new entity with custom fields using any of the entity types listed in “Base BillingCenter Data Objects” on page 158.

- Modify a small subset of the attributes of an existing base entity using overrides.
- Remove (or hide) an extension to a base entity that exists in the `extensions` folder as an `.etx` declaration file.
- Remove (or hide) a base entity that exists in the `extensions` folder as an `.eti` declaration file.

However, using extensions, you cannot:

- Delete a base entity or any of its fields. If you do not use a particular base entity or one of its fields, then simply ignore it.
- Change most of the attributes of a base entity or any of its fields.

## Strategies for Extending the Base Data Model

Extending the data model means one of the following:

- You want to add new fields to an existing entity.
- You want to create a new entity.

During planning for data model extensions, you need to consider performance implications. For example, if you add hundreds of extensions to a major object, this can conceivably exceed a reasonable row size in the database.

### Adding Fields to an Entity

If an entity has almost all the functionality you need to support your business case, you can add one or more fields to it. In this sense, Guidewire uses the term *field* to denote one of the following:

- Column
- Typekey
- Array
- Foreign key

See “Data Column and Field Types” on page 144 for a description of these fields.

### Subtyping a Non-Final Entity

If you want to find a new use for an existing entity, you can subtype and rename it. For instance, suppose that you want to track individuals who have already had the role of `IssueOwner`. In this case, it can be useful to create `PastIssueOwner`.

### Creating a New Entity

Occasionally, careful review of the base application data model makes it clear that you need to create a new entity. There are many types of base entities within Guidewire applications. However, Guidewire **strongly** recommends in general practice that you always use one of the following types if you create a new entity:

<code>retireable</code>	This type of entity is an extension of the <code>editable</code> entity. It is not possible to delete this entity. It is possible to retire it, however.
<code>versionable</code>	This type of entity has a version and an ID. It is possible to delete entities of this type from the database.

As a general rule, Guidewire recommends the following:

- Make the new entity `versionable` if it is not necessary for another entity to refer to the entity through the use of a foreign key.
- Make the entity `retirable` otherwise.

See “Data Entities and the Application Database” on page 153 for more information on these data types.

In general, you typically want to create a new entity under the following circumstances:

- If your business model requires an object that does not logically exist in the application. Or, if you have added too many fields to an existing entity, and want to abstract away some of it into a new, logical entity.
- If you need to manage arrays of objects, as opposed to multiple objects, you can create an entity array.

### Reference Entities

To store some unchanging reference data, such as a lookup table that seldom changes, you can create a reference entity. An example of a business case for a reference entity is a list of typical reserve amounts for a given exposure. To avoid the overhead of maintaining foreign keys, make reference entities keyable. Unless you want to build in the ability to edit this information from within the application, set `setterscriptability = hidden`. This prevents Gosu code from accidentally overwriting the data.

Guidewire recommends that you determine that this is not really a case for creating a typelist before you create a reference entity. See “Defining a Reference Entity” on page 216 for more information.

## What Happens If You Change the Data Model?

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

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

---

**WARNING** Do not directly modify the physical database that BillingCenter uses. Only make changes to the BillingCenter 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
- BillingCenter data model
- Field encryption
- Typelists

In addition to these generic changes, the following specific localization changes trigger a database upgrade:

- In file `localization.xml`, any change to the `<LinguisticSearchCollation>` subelement on the `<GWLocale>` element of the default application locale forces a database upgrade at application server startup.
- In file `collations.xml`, any change to the source definition of the `DBJavaClass` definition forces a database upgrade at application server startup.

## Naming Restrictions for Extensions

BillingCenter uses the names of extensions as the basis for several other internally-generated structures, such as database elements and Java classes. Because of this, it is important that you adhere to the naming requirements and guidelines described in this section.

**IMPORTANT** Deviations from these guidelines can result in product errors or unexpected behavior.

An extension name cannot start with a number; it must start with a letter. Other than that, an extension name can contain letters, numbers, or underscores (\_). Guidewire does not permit any other characters in extension names.

## Defining a New Data Entity

You define all new data entity objects in declaration files that end with the .eti extension. You do this through Guidewire Studio. Studio automatically manages the process and stores the .eti file in the correct location in the application (in the **Data Model Extensions → extensions** folder).

### To create a new entity

1. Create a file for that entity through Studio:
  - a. Navigate to **configuration → config → Extensions → Entity**.
  - b. Right-click **Entity**, and then click **New → Entity**.
2. In the **Entity** dialog, specify the entity definition values.  
Add fields to your new data entity. In the **Field** drop-down list, select the field type to add, and then click **Add** . For example:

XML tag	Use to add
<array>	An array of entities
<column>	A field with a simple data type
<foreignkey>	A field referencing another entity
<typekey>	A field with a typelist

See “Data Object Subelements” on page 174 for information on the possible XML elements that you can add to your new entity definition.

3. Deploy your changes to the application server. You must redeploy the application after you make any change to the Guidewire BillingCenter data model. See “Deploying Data Model Changes to the Application Server” on page 223 for details.

## Extending a Base Configuration Entity

You define all of your entity-type extensions in files that end with the .etx extension. You do this through Guidewire Studio. Studio automatically manages the process and stores the .etx file in the correct location in the application.

**IMPORTANT** Guidewire provides certain entity extensions as part of the base application configuration. Many of the extension index definitions address performance issues. Other extensions provide the ability to configure the data model in ways that would not be possible if the extension was part of the base data model. Do not simply overwrite a Guidewire extension with your own extension without understanding the full implications of the change.

BillingCenter 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 Account, User, Contact, and others are extendable. (You can determine if an entity is extendable by looking in the *Data Dictionary* to see if it supports the `Extendable` attribute. The *Data Dictionary* displays the list of attributes for that entity type directly underneath the entity name.)

Use the `<extension>` XML root element to create an extension entity. Before creating a new extension file, first determine if one already exists.

- If an extension file for the entity does, then edit that file to extend the entity.
- If an extension file for the entity does not exist, then create the new extension file and populate it accordingly.

Do **not** attempt to create multiple extension files for the same entity. You can reference a given existing entity in only **one** extension (.etx or .tx) file. If you attempt to extend (or define) the same entity in multiple files, then the BillingCenter application server generates an error at application start up. In all cases, Studio refuses to create entity or extension files with the same duplicate name.

### To create a new extension file

The simplest (and safest) way to create a new extension file is to let Studio manage the process.

1. In the Studio Project window, navigate to `configuration` → `config` → `Metadata` → `Entity`, and then locate the entity that you want to extend.
2. Right-click the entity, and then click `New` → `Entity Extension`. Studio creates a basically empty extension file named `<entity>.etx`, places it in the `configuration` → `config` → `Extensions` → `Entity` folder, and opens it in a view tab for editing.

**Note:** If an extension file for the selected entity file already exists, Studio does not permit you to create another one. If the file name in the Entity Extension dialog box is grayed out, that means that an extension already exists. In that case, search in the `configuration` → `config` → `Extensions` → `Entity` folder for an existing extension file.

3. Populate the extension with the required attributes.
4. Deploy your changes to the application server. You must redeploy the application after you make any change to the Guidewire BillingCenter data model. See “Deploying Data Model Changes to the Application Server” on page 223 for details.

## Working with Attribute Overrides

It is possible to override certain attribute values (fields) on entities that Guidewire defines in files to which you do not have direct access. For example, you do not have write access to any entity definition files in the `configuration → config → Metadata → Entity` subfolders. Guidewire provides a limited number of override elements for use in `.etx` extension files in the `configuration → config → Extentions` folder.

To use an override element:

- If an entity extension file (`.etx`) already exists in the **Extensions** folder, then add one of the specified override elements to the existing file.
- If an entity extension file (`.etx`) does not already exist in the **Extensions** folder, then you need to create one and add an override element to that file.
- If an entity definition file (`.eti`) exists in the **Extensions** folder, then you can modify the original field definition. You do not need to use an override element.

Only add override elements to `.etx` files in the `configuration → config → Extentions` folder. Do not attempt to add an override element to a file in any other folder or to any other file type.

The following list describes the attributes that you can override by using an override element in an `.etx` file in the **Extensions** folder:

Override element	Attributes that you can override
<code>&lt;array-override&gt;</code>	<code>triggersValidation</code>
<code>&lt;column-override&gt;</code>	<code>createhistogram</code> <code>default</code> <code>nullok</code> <code>size</code> <code>supportsLinguisticSearch</code> <code>type</code>
<code>&lt;foreignkey-override&gt;</code>	<code>nullok</code> <code>triggersValidation</code>
<code>&lt;onetoone-override&gt;</code>	<code>triggersValidation</code>
<code>&lt;typekey-override&gt;</code>	<code>default</code> <code>nullok</code>

These attributes have the following meanings:

Attribute	Description
<code>createhistogram</code>	Use to turn on (or off) the creation of a histogram during the generation of table statistics. This change does <b>not</b> 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>&lt;column&gt;</code> ” on page 178.
<code>default</code>	Use to change the default value given to the field (column) during new entity creation.
<code>nullok</code>	Use to make the <code>nullok</code> attribute to be more restrictive. You can <b>only</b> make it more restrictive, for example, changing it from <code>nullok="true"</code> to <code>nullok="false"</code> .
<code>size</code>	Use to change the size of a column. See “A size Attribute Example” on page 211.
<code>supportsLinguisticSearch</code>	Use to enable linguistic search on a column.
<code>triggersValidation</code>	Use to determine if BillingCenter initiates validation on changes to an array, a foreign key, or a one-to-one entity.
<code>type</code>	Use to change the data type of a column to a data type that is of a different value type. For example, suppose that you have a <code>String</code> column that currently is of <code>shorttext</code> and you want to make it use <code>longtext</code> . In this case, you use a <code>&lt;column-override&gt;</code> subelement to modify the original column definition.

## Overriding Data Type Attributes

Besides the attributes that you can specifically override using the `<column-override>` element, you can also modify data type attributes on a column. You do this through the use of nested `<columnParam>` subelements within the `<column-override>` element.

For example, the base configuration Contact entity defines a TaxID column (in `Contact.eti`):

```
<column createhistogram="true"
       desc="Tax ID for the contact (SSN or EIN)."
       name="TaxID"
       type="ssn"/>
```

To encrypt the contents of this column (a reasonable course of action), create a Contact extension (`Contact.etx`) and use the `<column-override>` element to set the `encryption` attribute on the column:

```
<column-override name="TaxID">
  <columnParam name="encryption" value="true"/>
</column-override>
```

### See also

- See “`<columnParam>` Subelement” on page 180 for a description of the `<columnParam>` element and the column attributes that you can modify using this element.

## A size Attribute Example

You can change the size of the Name column for a Document entity as follows:

1. Open Guidewire Studio.
2. Navigate to **configuration** → **config** → **Metadata** → **Entity**, right-click `Document.eti`, and then click **Create Extension File**.
3. Before the final `</extension>` tag, insert the following code to set the size of the Name column to 100:  

```
<column-override name="Name">
  <columnParam name="size" value="100"/>
</column-override>
```
4. Save the file.

## A triggersValidation Example

You use the `triggersValidation` attribute to instruct BillingCenter whether changes to an array, a foreign key, or a one-to-one entity initiates validation on that entity. To illustrate, in the base configuration, Guidewire defines the Account entity in file `Account.eti`.

```
<entity ... entity="Account" ...>
  ...
  <array arrayentity="UserRoleAssignment"
        desc="Role Assignments for this account."
        exportable="false"
        name="RoleAssignments"
        triggersValidation="true"/>
  ...
</entity>
```

The definition of the `RoleAssignments` array specifies that if any element of the array changes, the change triggers a validation of the object graph that includes the array. Suppose, for some reason, that you want to turn off validation even if changes occur to the `RoleAssignments` array. To do so, you need to create an extension file with an `<array-override>` element that modifies the `triggersValidation` attribute set on the base data object.

The following steps illustrate this concept.

### To override a triggersValidation attribute

1. Create an `Account.etx` file.
  - a. Find the `Account.eti` file in the Studio configuration tree. You can use **Ctrl+N** to find the file.

**b.** Select the file, right-click and click **New → Entity Extension**.

Studio creates an `Account.etc` file and places it in the `configuration → config → Extentions → Entity` folder.

**2.** Populate `Account.etc` with the following:

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="Account">
  <array-override name="RoleAssignments" triggersValidation="false">
  </extension>
```

**3.** Stop and restart the application server. The application server recognizes that there are changes to the data model and automatically runs the upgrade utility on start up.

This effectively switches off the validation that usually occurs on changes to elements of the `RoleAssignments` array.

## Extending the Base Data Model: Examples

As described in “Defining a New Data Entity” on page 208, you can define entirely new custom entities that become part of the BillingCenter 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. BillingCenter 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 BillingCenter. 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>&lt;delegate&gt;</code> element.
Step 2: Define the Delegate Functionality	Create a Gosu enhancement to provide any functionality that you want to expose on your delegate.

Task	Description
Step 3: Add the Delegate to the Parent Entity	Use the <implementsEntity> element to associate the delegate with the parent entity.
Step 4: Deploy your Data Model Changes	Deploy your data model changes. You may need to regenerate any Java API file or web service WSDL files after data model changes.

The following topics describe this process.

### Step 1: Create the Delegate Object

The first step in defining a new delegate is to create the delegate file and populate it with the necessary code.

#### To create a delegate object

1. Within Guidewire Studio, navigate to **configuration** → **config** → **Extentions** → **Entity**.
2. Right-click and click **New** → **Entity Extension**.
3. Enter the file name, using the name of the delegate and adding the **.eti** extension. This action creates an empty file. You use this file to define the fields on the delegate.
4. Enter the delegate definition in the delegate file. If necessary, find an existing delegate file and use it as a model for the syntax.

For example, in the base configuration, Guidewire defines the implementation of the **Assignable** delegate as follows:

```
<delegate ... name="Assignable">
  ...
  <column desc="Time when entity last assigned"
    exportable="false"
    name="AssignmentDate"
    setterScriptability="hidden"
    type="datetime"/>
  <column desc="Date and time when this entity was closed. (Not applicable to all assignable entities)"
    exportable="false"
    name="CloseDate"
    type="datetime"/>
  <foreignkey columnName="AssignedGroupID"
    desc="Group to which this entity is assigned; null if none assigned"
    exportable="false"
    fkentity="Group"
    name="AssignedGroup"
    setterScriptability="ui"/>
  ...
</delegate>
```

### Step 2: Define the Delegate Functionality

Next, you need to provide functionality for the delegate. While there are several ways to do this, you must use a Gosu enhancement implementation.

Java class implementation	In the base configuration, Guidewire provides a Java class implementation for each delegate to provide the necessary functionality. The <b>Delegate</b> object designates the Java class through the <b>javaClass</b> attribute. It is not possible for you to create and use a Java class for this purpose.
Gosu enhancement implementation	You must implement the delegate functionality through a Gosu enhancement that defines any functionality associated with the fields on the delegate. By providing the name of the delegate entity to the enhancement as you create it, you inform Studio that you are adding functionality for that particular delegate. Studio automatically recognizes that you are enhancing the delegate.

### Step 3: Add the Delegate to the Parent Entity

The next step is to associate a delegate with an entity using the `<implementsEntity>` element in the entity definition.

- If you are creating a *new* entity, then you need to add the `<implementsEntity>` element to the entity definition .eti file.
- If you are working with an *existing* entity, then you need to add the `<implementsEntity>` element to the entity extension .etx file.

The following steps illustrate this process by creating a new entity. The steps to extend an existing entity are similar.

#### To associate a delegate with a new entity

1. Within Guidewire Studio, navigate to **configuration** → **config** → **Extentions** → **Entity**.
  2. Right-click and select **New** → **Entity Extension** from the submenu.
  3. Enter the name of the entity file. You must add the .eti extension. Studio does not do this for you. This action creates an empty file. You use this file to associate the delegate with your entity. If necessary, find an existing entity file and use it as a model for the syntax.
- Note:** Guidewire recommends that you add either the Ext prefix or suffix to all entities that you create or extend. If you do so, do so consistently. Always use prefixes or always use suffixes.
4. Enter the necessary text in this file, using the `<implementsEntity>` element to specify the delegate. For example (in the ClaimCenter base configuration), Guidewire defines the `Claim` entity—in `Claim.eti`—so that it implements a number of delegates, including the `Assignable` and `Validatable` delegates. The definition looks like this:

```
<entity xmlns="http://guidewire.com/datamodel" ... entity="Claim" ... />
<implementsEntity name="Validatable"/>
<implementsEntity name="Assignable"/>
...
...
```

### Step 4: Deploy your Data Model Changes

After completing these steps, you need to deploy your data model changes. If necessary, see “Deploying Data Model Changes to the Application Server” on page 223 for details. Depending on whether you are working in a development or production environment, you need to perform different tasks. You may need to regenerate any Java API file or web service WSDL files after data model changes.

## Extending a Delegate Object

**Note:** A Delegate data object is a reusable entity that contains an interface and a default implementation of that interface. See “Delegate Data Objects” on page 158 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 BillingCenter:

- `<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 BillingCenter, you can extend the system delegate `AddressAutofillable`.

You can only extend a delegate if the base configuration definition file for that delegate contains the following:  
`extendable="true"`

The default for the `extendable` attribute on `<delegate>` is `false`. Therefore, if it is not set explicitly to `true` in the delegate definition file, you cannot extend that delegate.

Do not attempt to change the graph to which a Guidewire base entity belongs through extension. In other words, do not attempt to change the delegate that a Guidewire base entity implements through an extension entity using `<implementsEntity>`. BillingCenter generates an error if you attempt to do so.

#### To extend a delegate object

1. Navigate to **configuration** → **config** → **Extentions** → **Entity**.
2. Right-click and select **New** → **Entity Extension**.
3. Enter the name of the delegate that you want to extend and add the `.etx` extension. Studio opens an empty file.
4. Enter the delegate definition in the delegate extension file. If necessary, find an existing delegate file and use it as a model for the syntax. For details, see “Creating a New Delegate Object” on page 212.

#### See also

- “The BillingCenter Data Model” on page 147
- “Delegate Data Objects” on page 158
- “`<implementsEntity>`” on page 188
- “Creating a New Delegate Object” on page 212

## 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.

BillingCenter 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. BillingCenter 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 BillingCenter bases the subtype entity). For a description of required and optional attributes, see “Subtype Data Objects” on page 169.

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 174.

#### Example

This example defines an `Inspector` entity as a subtype of `Person`. The `Inspector` entity includes a field for the inspector’s license. To create the `InspectorExt.eti` file, navigate to the **Extensions** folder, then select **New** → **Entity Extension** from the right-click submenu. Enter the full name including the extension in the dialog.

```
<?xml version="1.0"?>
<subtype xmlns="http://guidewire.com/datamodel" desc="Professional inspector" displayName="Inspector"
          entity="InspectorExt"
          supertype="Person">
    <column name="InspectorLicenseExt" type="varchar" desc="Inspector's business license number">
        <columnParam name="size" value="30"/>
    </column>
</subtype>
```

Notice that while `InspectorExt` is subtype of `Person`, `Person`, itself, is a subtype of `Contact`. BillingCenter automatically adds the new `InspectorExt` type to the `Contact` typelist. This is true, even though BillingCenter marks the `Contact` typelist as `final`.

To see this change:

- To see this change in the *BillingCenter Data Dictionary*, you must restart the application server.
- To see this change in the `Contact` typelist in Studio, you must restart Studio.

## Defining a Reference Entity

You use a reference entity to store reference data for later access from within BillingCenter 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

You can populate a reference entity by importing its data, and then you can query it using Gosu expressions. If you do not want BillingCenter 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 BillingCenter 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. BillingCenter 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 149
defining a new entity	"Defining a New Data Entity" on page 208
defining an array field	"<array>" on page 176
defining a foreign key field	"<foreignkey>" on page 186

### Example

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

- 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 `Account.etcx`.

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

```
<entity entity="ExampleRetireableArrayEntityExt" table="exampleretarray" ... >
    ...
    <foreignkey name="AccountID" fkentity="Account" desc="FK back to Account" 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 BillingCenter interface. (See “View Entity Data Objects” on page 171 for details.) Some default PCF pages make use of list view entities and some do not. If you add a new field to an entity, then you need to decide if you want to extend a **viewEntity** to include this new field. This can potentially avoid performance degradation.

The following example illustrates a case in which you add an extension both to a primary entity and its corresponding **viewEntity**. First search for **Activity.etc** to determine if one exists. (Use Ctrl+N to open the search dialog.)

Add the following to **Activity.etc**:

```
<extension entityName="Activity">
    ...
    <column type="bit"
        name="validExt"
        default="true"
        nullok="true"
        desc="Sample bit extension, with a default value."/>
    ...
</extension>
```

Next, search for **ActivityDesktopView.etc**. Suppose that you do not find this file, but you see that **ActivityDesktopView.eti** exists. As this is part of the base configuration, you cannot modify this declaration file. However, find the highlighted file in Studio and select **New → Entity Extension** from the right-click submenu. This opens a mostly blank file.

Enter the following in **ActivityDesktopView.etc**:

```
<viewEntityExtension entityName="ActivityDesktopView">
```

```
<viewEntityColumn name="validExt" path="validExt"/>
</viewEntityExtension>
```

**Note:** The path attribute is always relative to the primary entity on which you base the view.

These data model changes add a `validExt` column (field) to the `Activity` object, which is also accessible from the `ActivityDesktopView` entity.

### Extending an Existing View Entity with a Currency Column

If you create or extend a view entity that references a column that is of `type="currencyamount"`, you must handle the view entity extension in a particular manner. If the view entity extension references a `currencyamount` column, then you also need to define the `currencyProperty` that the original column definition specifies on the view entity as well.

Suppose, for example, that in ClaimCenter, you extend the `PriorClaimView` view entity by adding an `OpenReserves` field that has a path reference to `ClaimRpt.OpenReserves`:

```
<viewEntityColumn name="OpenReserves" path="ClaimRpt.OpenReserves"/>
```

Looking at the definition of `ClaimRpt`, you see the following:

```
<column default="0" desc="The open reserves." name="OpenReserves" nullok="false" type="currencyamount">
  <columnParam name="currencyProperty" value="ClaimCurrency"/>
</column>
```

Notice that `ClaimRpt.OpenReserves` is of `type="currencyamount"`. Thus, if you extend `PriorClaimView` with this field as indicated, you also need to add the following to `PriorClaimView.etx`:

```
<viewEntityTypekey name="ClaimCurrency" path="Currency"/>
```

By defining a `ClaimCurrency` column on the view entity, the view entity loads the claim currency as a part of the view entity. This makes the claim currency available to the `currrencyamount` column and ClaimCenter avoids loading the whole entity while dereferencing `Claim.Currency`.

## Removing Objects from the Base Configuration Data Model

It is possible to safely remove certain objects from the base configuration data model. You can do this only if the data object declaration file exists in the `Data Model Extensions → extensions` folder, either as an `.eti` file or an `.etx` file.

The following table lists the objects that you can remove (or hide) in the base configuration:

Object to remove	Location	File	See
Base configuration <code>entity</code>	extensions	<code>.eti</code>	"Removing a Base Extension Entity" on page 220
Base configuration <code>extension</code>	extensions	<code>.etx</code>	"Removing an Extension to a Base Object" on page 221

Guidewire recommends that you review the material in “Implications of Modifying the Data Model” on page 221 before you remove an object from the data model.

**IMPORTANT** Guidewire provides certain entity extensions as part of the base application configuration. Many of the extension index definitions address performance issues. Other extensions provide the ability to configure the data model in ways that would not be possible if the extension was part of the base data model. Do not modify a Guidewire extension without understanding the full implications of the change.

**WARNING** Do not attempt to remove a base configuration data object (meaning one defined in the **Data Model Extensions → metadata** folder). Also, do not attempt to remove any extension marked as internal. Any attempt to do so can invalidate your Guidewire installation, causing the application server to refuse to start.

## Removing a Base Extension Entity

It is possible to remove an extension entity that is part of the base data model. You can only remove an extension *entity* that the base configuration defines in the **configuration → config → Extentions → Entity** folder as an .eti file.

For example, in PolicyCenter, the base configuration includes a number of *entity extension* files in the **Extensions** folder, including:

- RateGLClassCodeExt
- RateWCClassCodeExt
- ...

There are two ways to remove an extension entity from the **Extensions** folder:

- For .eti files that Guidewire added as part of the base configuration, you need to edit the file, remove the current content, and insert a <deleteEntity> element in its place.
- For .eti files that you added as part of your customization process, you need merely delete the file.

In actual practice, you are not removing or deleting either the physical file or the extension itself. You are merely hiding—or negating—the effects of the extension entity in the data model.

### To delete a base extension entity

1. Open the entity extension .eti file. This file must be located in the **extensions** folder.
  - If the .eti file is one that you created (meaning it is not part of the Guidewire-provided base configuration), then you merely need to delete the file. You can then omit the next step and continue to step 3.
  - If the .eti file is part of the Guidewire-provided base configuration, then continue to the next step.
2. Use the <deleteEntity> object to define the extension entity to remove from the data model. For example, if you want to remove an extension entity named RateGLClassCodeExt, then enter the following:

```
<?xml version="1.0"?>
<deleteEntity xmlns="http://guidewire.com/datamodel" name="RateGLClassCodeExt" />
```
3. Stop and restart the application server. At start up, the application server recognizes a data model change and automatically upgrades the database.

If you encounter error messages, or the application server refuses to start, examine your code and correct any issues before you attempt to continue.

## Removing an Extension to a Base Object

It is also possible to remove an extension to a base data model object. You can only remove an entity *extension* that the base configuration defines in the **configuration → config → Extensions → Entity** folder as an .etx file.

As with the case with extension entities in the **Extensions** folder, there are two ways to handle the removal of entity extensions:

- For .etx files that Guidewire added as part of the base configuration, you need to edit the file, remove the current content, and insert a <deleteEntity> element in its place.
- For .etx files that you added as part of your customization process, you need merely delete the file.

**IMPORTANT** You cannot delete an extension marked as internal. Any attempt to do so can invalidate your Guidewire installation, causing the application server to refuse to start.

### To remove a base extension

1. Navigate to the **extensions** folder and open the declaration file for the entity extension that you want to remove.
  - If the .etx file is one that you created (meaning it is not part of the Guidewire-provided base configuration), then you merely need to delete the file. You can then omit the next step and continue to step 3.
  - If the .etx file is part of the Guidewire-provided base configuration, then continue to the next step. For example, suppose that you want to remove (hide) the extension defined in the base configuration for the **Contact** entity. In that case, you open **Contact.etx** in the **Extensions** folder.
2. Delete the contents of the declaration file and insert a blank skeleton definition. For example, for the **Contact** extension, use the following:

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="Contact"/>
```
3. Stop and restart the application server. At start up, the application server recognizes a data model change and automatically upgrades the database.

If you encounter error messages, or the application server refuses to start, examine your code and correct any problems before you attempt to continue.

## Implications of Modifying the Data Model

Any change to a data object modifies the underlying BillingCenter database. Typically, each data entity has a corresponding table in the database and each object attribute maps to a table column. If you remove or alter a data object, the possibility exists that your object contains data such as rows in an entity table or data in a column.

This topic covers the following:

- Does Removing an Extension Make Sense?
- Writing SQL for Extension Removal
- Strategies for Handling Extension Removal
- Troubleshooting Modifications to the Data Model

### Does Removing an Extension Make Sense?

Typically, removing a data object only makes sense in your development environment. If you build a new configuration, it can sometimes be necessary to remove an object rather than to drop it and to recreate the database. Dropping the database destroys any data that currently exists. This might not be an option if you share a database instance with multiple developers. In this case, removing the object is less painful for the development team.

During server start up, BillingCenter checks for configuration changes, such as modified extensions, that require a database upgrade. Until the database reflects the underlying configuration, BillingCenter 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. BillingCenter refuses to start if there are `null` values in a non-nullable column.
- Change the underlying data type of a column, for example, changing a `varchar` column to `clob` or `varchar` column to `int`.
- Shorten the length of a `varchar/text-based` column (for example, `mediumtext` to `shorttext`) if this truncates data in the column. If shortening the length does not require truncating existing data, the upgrader can handle both shortening the length of a `varchar` column and increasing the length of a `varchar` column. (It can increase the length up to 8000 characters for SQL Server.)

### Writing SQL for Extension Removal

Some modifications to the data model can require that you write an SQL statement to synchronize the database with the data model. How complex this SQL depends on what you want to remove. For example, to remove a field on an object, you need to alter the table and drop the column. However, if your extension includes foreign keys or indexes, then you need to take into account the referential integrity rules for the database—and your SQL becomes correspondingly more complex.

In a development environment, you can use the trial-and-error approach to writing your SQL.

In a production environment, in which—typically—there is data to preserve in each extension, the SQL can require an additional layer of complexity. For example, if you write an SQL statement in which a column type changes, your SQL can do something similar to the following:

- The SQL creates a temporary column.
- It copies data from the existing extension column to the temporary column.
- It drops the existing extension column.
- It recreates the extension column with its new properties as appropriate.
- It copies the data from the temporary column to the newly recreated column.
- It removes the temporary column.

However, in most cases, this is not necessary as Guidewire provides version triggers that modify the database automatically if the application detects data model changes. You only need to do manual SQL modification of the database if you want to modify your own extensions. Even in that case, Guidewire strongly recommends that a database administrator (DBA) always develop the SQL to use in removing an extension.

**WARNING** Be very careful of making changes to the data model on a live production database. You can invalidate your installation.

### Strategies for Handling Extension Removal

Suppose that you have a development environment with multiple developers all using the same database instance. Before modifying the data model, first you need to communicate with your team to make them aware of what you plan to do. A good way to communicate your intentions is to provide the team with the SQL you intend to execute along with a list of impacted references files. After communicating with your team, follow a process similar to the following if removing a data object:

1. Remove the extension entity or entity extension using the methods outlined in the following sections:

- “Removing a Base Extension Entity” on page 220
  - “Removing an Extension to a Base Object” on page 221
2. Remove any references to the object in other parts of your configuration. If you do not remove these references, BillingCenter displays error messages during server start-up.
3. Check in your changes.
4. Open an SQL command line appropriate to your server. For example, if you use Microsoft SQL Server, then open a query through the SQL Enterprise Manager.
5. Run your SQL statement to remove your extension.
6. Regenerate the toolkit.

In a production environment, Guidewire recommends that you include formal testing and quality assurance before removing or modifying an extension. Also, involve your company database administrator (DBA) and any impacted departments. Guidewire recommends also that you document your change and the reasons for it.

### Troubleshooting Modifications to the Data Model

It is possible to change an `integer` column to a `typekey` column (and the reverse). However, `integer` values in the database do not necessarily map to a valid ID within the referenced typelist table after you make this type of change. Related to this, removing typecodes from a typelist (instead of retiring them) can cause data inconsistencies as well. If you have data that references a non-existent typecode, the upgrade does not complete and the server refuses to start. Instead of removing typecodes, retire them instead.

You can remove an extension field or the entire entity from the data model. If you do this, the server logs an informational message to the console such as:

```
bcx_ex_ProviderServicedStates: mismatch in number of columns - 5 in data model, 6 in physical database
```

## Deploying Data Model Changes to the Application Server

How your deploy changes to the data model depends on if you are working in a *development* or *production* environment.

### Development Environment

If you are working in a development environment, then do the following:

1. Use the following command (from the application `bin` directory) to regenerate the *Data Dictionary* so that it reflects your data model changes:

```
gwbc 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), BillingCenter runs the database upgrade tool during application start up.

### Production Environment

If you are working in a production environment, then do the following:

1. Use the following command (from the application `bin` directory) to regenerate the *Data Dictionary* so that it reflects your data model changes:

```
gwbc regen-dictionary
```

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

See the “Key BillingCenter gwbc Commands” on page 57 in the *Installation Guide* for information on how to use these commands.

3. Copy this file to the application server. The target location of the file is dependent on the application server. If necessary (and it is almost always necessary if you change the data model), BillingCenter runs the database upgrade tool during application start up.

# Data Types

This topic describes the Guidewire data types, what they are, how to customize a data type, and how to create a new data type.

This topic includes:

- “Overview of Data Types” on page 225
- “The Data Types Configuration File” on page 228
- “Customizing Base Configuration Data Types” on page 229
- “Working with the Medium Text Data Type (Oracle)” on page 231
- “The Data Type API” on page 231
- “Defining a New Data Type: Required Steps” on page 233
- “Defining a New Tax Identification Number Data Type” on page 233

**See also**

- “Monetary Amounts in the Data Model and in Gosu” on page 115 in the *Globalization Guide*

## Overview of Data Types

In the Guidewire data model, a *data type* is an augmentation of an object property, along three axes:

Axis	Description
Constraint	A data type can restrict the range of allowable values. For example, a String data type can restrict values to a maximum character limit.
Persistence	A data type can specify how BillingCenter stores a value in the database and in the object layer. For example, one String data type can store values as CLOB (Character Large Object) objects. Another String data type can store values as VARCHAR objects.
Presentation	A data type can specify how the BillingCenter interface treats a value. For example, a String data type can specify an input mask to use in assisting the user with data entry.

Guidewire stores the definitions for the base configuration data types in \*.dti files in the datatypes directory. Each file corresponds to a separate data type, which the file name specifies.

Every data type has an associated Java or Gosu type (defined in the valueType attribute). For example, the associated type for the datetime data type is java.util.Date. Thus, you see the following XML code in the datetime.dti file.

```
<DataTypeDef xmlns="http://guidewire.com/datatype"
    type="com.guidewire.pl.metadata.datatype2.impl.DateTimeTypeDef"
    valueType="java.util.Date">
    ...

```

In a similar manner, the decimal data type has an associated type of java.math.BigDecimal.

```
<DataTypeDef xmlns="http://guidewire.com/datatype"
    type="com.guidewire.pl.metadata.datatype2.impl.DecimalTypeDef"
    valueType="java.math.BigDecimal">
    ...

```

## Working with Data Types

In working with data types, you can do the following:

Operation	Description
Customize an existing data type	Modify the data type definition in file datatypes.xml, which you access through Studio. You can modify only a select subset of the base configuration data types. See “Customizing Base Configuration Data Types” on page 229.
Create a new data type	Create a .dti definition file and place it in <i>BillingCenter/modules/configuration/config/datatypes</i> . You also need to create Gosu code to manage the data type. See “Defining a New Data Type: Required Steps” on page 233.
Override the data type on a column	Override the parameterization of the data type on individual columns (fields) on an entity. For example, you can make a VARCHAR column in the base data model use encryption by extending the entity and setting the encryption parameter on a <columnParam> element.

## Using Data Types

You can use any of the data types for data fields (except for those that Guidewire reserves for itself). This includes data types that are part of the base configuration or data types that you create yourself. If you add a new column (field) to an entity or create a new entity, then you can use any data type that you want for that entity field. You do this by setting the type attribute on the column. For example:

```
<extension entityName="Account">
    <column name="NewCompanyName" type="CompanyName" nullok="true" desc="Name for the new company."/>
</extension>
```

If you add too many large fields to any one table, you can easily reach the maximum row size of a table. In particular, this is a problem if you add a large number of long text or VARCHAR fields. Have your company database administrator (DBA) determine the maximum row size and increase the page size, if needed.

### Guidewire-Reserved Data Types

Guidewire reserves the right to use the following data types exclusively. Guidewire does not support the use of these data types except for its own internal purposes. Do not attempt to create or extend an entity using one of the following data types:

- foreignkey
- key
- typekey
- typelistkey

## Database Data Types

Guidewire bases its base configuration data types on the following database data types:

- BIT
- BLOB
- CLOB
- DECIMAL
- INTEGER
- TIMESTAMP
- VARCHAR

## Data Types and Database Vendors

It is possible to see both VARCHAR and varchar in the Guidewire documentation. This usage has the following meanings.

### All Upper-case Characters

This refers to database data types generally, for example VARCHAR and CLOB (Character Large Object). Of the supported database vendors, the Oracle (and H2) databases use upper-case data type names, while the SQL Server database uses lower-case data type names. To view the entire set of database data types, consult the database vendor's documentation.

### All Lower-case Characters

This refers to Guidewire data types generally, for example, varchar and text. You can determine the set of Guidewire data types by viewing the names of the data type metadata definition files (\*.dti) in the following application locations:

config/datatypes

## Defining a Data Type for a Property

Guidewire associates data types with object properties using the following annotation:

`gw.datatype.annotation.DataType`

The annotation requires you to provide the name of the data type, along with any parameters that you want to supply to the data type.

- You associate a data type with a metadata property by specifying the `type` attribute on the `<column>` element.
- You specify any parameters for the data type with `<columnParam>` elements, children of the `<column>` element.

At runtime, BillingCenter translates these metadata elements into instances of the `gw.datatype.annotation.DataType` annotation on the property corresponding to the `<column>`.

Each data type has a value type. You can associate a data type only with a property that has a feature type that matches the data type of the value type. For example, you can only associate a `String` data type with `String` properties.

**Note:** Guidewire BillingCenter does not enforce this restriction at compile time. (However, BillingCenter does check for any exception to this restriction at application server start up.) Guidewire permits annotations on any allowed feature, as long as you supply the parameters that the annotation requires. Therefore, you need to be aware of this restriction and enforce it yourself.

## The Data Types Configuration File

**IMPORTANT** You must perform a database upgrade if you make changes to the `datatypes.xml` file. You must increment the version number in `extensions.properties` (in BillingCenter Studio) to force a database upgrade upon application server start-up.

BillingCenter lets you modify certain attributes on a subset of the base configuration data types by using the `datatypes.xml` configuration file. You can access this file in Studio from `configuration → config → fieldvalidators`. You can modify the values of certain attributes in this file to customize how these data types work in BillingCenter.

This `datatypes.xml` file contains the following elements:

XML element	Description
<code>&lt;DataTypes&gt;</code>	Top XML element for the <code>datatypes.xml</code> file.
<code>&lt;...DataType&gt;</code>	Subelement that defines a specific <i>customizable data type</i> (for example, <code>PhoneDataType</code> , <code>YearDataType</code> , <code>MoneyDataType</code> ) and assigns one or more default values to each one.

**WARNING** Modify the `datatypes.xml` file with caution. If you modify the file incorrectly, you can invalidate your BillingCenter installation.

### `<...DataType>`

The `<...DataType>` element is the basic element of the `datatypes.xml` file. It assigns default values to base configuration data types that Guidewire permits you to customize. This element starts with the specific data type name. For example, the element for the `PercentageDec` data type in the `datatypes.xml` file is `<PercentageDecDataType>`.

The `<...DataType>` element has the following attributes:

Attribute	Description
<code>length</code>	Assigns the maximum character length of the data type.
<code>validator</code>	Binds the data type to a given validator definition. It must match the <code>name</code> attribute of the validator definition.
<code>precision</code>	Used for <code>DECIMAL</code> types only.
<code>scale</code>	<ul style="list-style-type: none"> <li>• <code>precision</code> is the total number of digits in the number.</li> <li>• <code>scale</code> is the number of digits to the right of the decimal point. The default value is 2.</li> </ul> <p>The value of <code>scale</code> must be less than the value of <code>precision</code>.</p> <p>For more information, see “The Precision and Scale Attributes” on page 229.</p>
<code>appscale</code>	Optional attribute for use with <code>money</code> data types.
	For more information, see “The Money Data Type” on page 231.

### Deploying Modifications to Data Types Configuration File

If you change the `datatypes.xml` file, then you need to deploy those changes to the application server. Most modifications to the `datatypes.xml` file take effect the next time the server reboots.

- BillingCenter reloads the `validator` attribute for data type definitions upon server reboot. This is so that you can rebind different validators to data types.

- BillingCenter does not reload other data type attributes such as `length`, `precision`, and `scale`. This is because BillingCenter applies these attributes only during the initial server boot. (It uses them during table creation in the database.) BillingCenter ignores any changes to these attributes unless something triggers a database upgrade. For example, if you modify a base entity, then BillingCenter triggers a database upgrade at the next server restart.

### Guidewire Recommendations for Modifying Data Types

Guidewire recommends the following:

- Make modifications to the data types before creating the BillingCenter database for the first time.
- Make modifications to the data types before performing a database upgrade that creates a new extension column.

BillingCenter looks at the data type definitions only at the time it creates a database column. Thus, it ignores any changes after that point. However, any differences between the type definition and the actual database column can cause upgrade errors or failure warnings. Therefore, Guidewire recommends that you exercise extreme caution in making changes to type definitions.

## Customizing Base Configuration Data Types

You can customize the behavior of the data types listed in `datatypes.xml`. To see exactly what you can customize for each data type, see “List of Customizable Data Types” on page 230. In general, though, you can customize some or all of the following attributes on a listed data type (depending on the data type):

- `length`
- `precision`
- `scale`
- `validator`

### The Length Attribute

Data types based on the `VARCHAR` data type have a `length` attribute that you can customize. This attribute sets the maximum allowable character length for the field (column).

### The Precision and Scale Attributes

Data types based on the `DECIMAL` data type have `precision` and `scale` attributes that you can customize. These attributes determine the size of the decimal. The `precision` value sets the total number of digits in the number and the `scale` value is the number of digits to the right of the decimal point.

There are special requirements for these attributes in working with monetary amounts. For more information, see “Precision and Scale of Monetary Amounts” on page 116 in the *Globalization Guide*.

### The Validator Attribute

Most data types have a `validator` attribute that you can customize. This attribute binds the data type to a given validator definition. For example, `PhoneDataType` (defined in `datatypes.xml`) binds to the `Phone` validator by its `validator` attribute. This matches the `name` attribute of a `<ValidatorDef>` definition in file `fieldvalidators.xml`.

```
//File datatypes.xml
<DataTypes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xsi:noNamespaceSchemaLocation="../../../../../../platform/p1/xsd/datatypes.xsd">
    ...
    <PhoneDataType length="30" validator="Phone"/>
    ...
</DataTypes>

//File fieldvalidators.xml
<FieldValidators>
```

```

...
<ValidatorDef description="Validator.Phone" input-mask="###-###-### x###" name="Phone"
    value="[0-9]{3}-[0-9]{3}-[0-9]{4}( x[0-9]{0,4})?" />
...
</FieldValidators>
```

**See also**

- For information on field validators in general, see “Field Validation” on page 239.
- For information on how to localize field validation, see “Configuring National Field Validation” on page 167 in the *Globalization Guide*.

## List of Customizable Data Types

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

BillingCenter 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` must be greater than or equal to `scale`.

Guidewire data type	Built on	Customizable attributes
ABContactMatchSetKey	VARCHAR	length
Account	VARCHAR	length, validator
AddressLine	VARCHAR	length, validator
ClaimNumber	VARCHAR	length, validator
CompanyName	VARCHAR	length, validator
ContactIdentifier	VARCHAR	length, validator
CreditCardNumber	VARCHAR	length, validator
DaysWorkedWeek	DECIMAL	precision, validator
DriverLicense	VARCHAR	length, validator
DunAndBradstreetNumber	VARCHAR	length, validator
EmploymentClassification	VARCHAR	length, validator
ExchangeRate	DECIMAL	precision, validator
Exmod	DECIMAL	precision, validator
FirstName	VARCHAR	length, validator
HoursWorkedDay	DECIMAL	precision, validator
LastName	VARCHAR	length, validator
MediumText	VARCHAR	length
Money	DECIMAL	precision, app, validator
PercentageDec	DECIMAL	precision
Phone	VARCHAR	length, validator
PolicyNumber	VARCHAR	length, validator
PostalCode	VARCHAR	length, validator
ProrationFactor	DECIMAL	precision, validator
Rate	DECIMAL	precision, validator
RatingLineBasisAmount	DECIMAL	precision, validator
Risk	DECIMAL	precision, validator
Speed	INTEGER	validator
SSN	VARCHAR	length, validator

Guidewire data type	Built on	Customizable attributes
VIN	VARCHAR	length, validator
Year	INTEGER	validator

### The Percentage Decimal Data Type

Guidewire builds the PercentageDec data type on top of the DECIMAL (3,0) data type. Only use decimal values from 0 to 100 inclusive.

### The Money Data Type

Guidewire provides the Money data type as the basis for the <monetaryamount> subelement in metadata definition files. The <monetaryamount> subelement is a compound field type, with a Money data type as one component and a typekey to the Currency typelist as the other component.

For more information, see “Monetary Amounts in the Data Model and in Gosu” on page 115 in the *Globalization Guide*.

## Working with the Medium Text Data Type (Oracle)

In working with the MEDIUMTEXT data type, take extra care if you use multi-byte characters, excluding CLOB-based data types such as LONGTEXT, TEXT, or CLOB in the Oracle database. (CLOB stands for Character Large OBject.) On Oracle, Guidewire supports any single-byte character set, or the multi-byte character sets UTF8 and AL32UTF8.

Oracle has a maximum column width, for non-LOB columns, of 4000 bytes. Thus, with a single-byte character set, you can store up to 4000 characters in a single column (because one character requires one byte). However, with a multi-byte character set, you can store fewer characters, depending on the ratio of bytes to characters for that character set. For UTF8, the ratio is at most three-to-one, so you can always safely store up to  $4000 / 3 = 1333$  characters in a single column.

Thus, Guidewire recommends:

- Limit the number of characters to 4000 if using a single-byte character set.
- Limit the number of characters to 1333 if using UTF8 or AL32UTF8. However, it is possible that some AL32UTF8 characters can be four bytes, and thus 1333 of them can potentially overflow 4000 bytes.

## The Data Type API

The classes in `gw.datatype` form the core of the Data Type API. Most of the time, you do not need to use data types directly, as Guidewire uses these internally in the system. However, there can be cases in which you need to access a data type, typically to determine the constraints information.

This topic includes:

- Retrieving the Data Type for a Property
- Retrieving a Particular Data Type in Gosu
- Retrieving a Data Type Reflectively
- Using the `IDataType` Methods

## Retrieving the Data Type for a Property

To retrieve the data type for a property, you could look up the annotation on the property. You could then look up the data type reflectively, using the `name` and `parameters` properties of the annotation. However, this is a cumbersome process. As a convenience, use the following method instead:

```
gw.datatype.DataTypes.get(gw.lang.reflect.IAnnotatedFeatureInfo)
```

For example:

```
var property = Claim.Type.TypeInfo.getProperty("ClaimNumber")
var claimNumberDataType = DataTypes.get(property)
```

The `gw.datatype.DataTypes.get(gw.lang.reflect.IAnnotatedFeatureInfo)` method also provides some performance optimizations. Therefore, Guidewire recommends that you use this method rather than looking up the annotation directly from the property.

## Retrieving a Particular Data Type in Gosu

If you need an instance of a particular data type, use the corresponding method on `gw.datatype.DataTypes`. A static method exists on this type for each data type in the system. Some data types have two methods:

- One method that takes all parameters
- One method that takes only the required parameters

For example:

```
var varcharDataType = DataTypes.varchar(10)
var encryptedVarcharDataType = DataTypes.varchar(10,
    /* validator */ null,
    logicalSize /* null,
    /* encryption */ true,
    /* trimwhitespace */ null)
```

## Retrieving a Data Type Reflectively

In rare cases, you may need to look up a data type reflectively. To do this, you need the name of the data type, and a map containing the parameters for the data type. For example:

```
var varcharDataType = DataTypes.get("varchar", { "size" -> "10" })
```

## Using the `IDataType` Methods

After you have a data type, you can access its various aspects using one of the `asXXXDataType` methods, which are:

- `asConstrainedDataType()` : `IConstrainedDataType`
- `asPersistentDataType()` : `IPersistentDataType`
- `asPresentableDataType()` : `IPresentableDataType`

For example, suppose that you want to determine the maximum length of a property:

```
var claim : Claim = ...
var claimNumberProperty = Claim.Type.TypeInfo.getProperty("ClaimNumber")
var claimNumberDataType = DataTypes.get(claimNumberProperty)
var maxLength = claimNumberDataType.asConstrainedDataType().getLength(claim, claimNumberProperty)
```

It may seem odd that the `getLength(java.lang.Object, gw.lang.reflect.IPropertyInfo)` method (in this example) takes the claim and the claim number property. The reason for this is that the constraint and presentation aspects of data types are dynamic, meaning that they are based on context.

Many of the methods on `gw.datatype.IConstrainedDataType` and `gw.datatype.IPresentableDataType` take a context object, representing the owner of the property with the data type, along with the property in question. This allows the implementation to provide different behavior, based on the context. If you do not have the context object or property, then you can pass `null` for either of these arguments.

If you implement a data type, then you must handle the case in which the context is unknown.

## Defining a New Data Type: Required Steps

The process of defining a new data type requires multiple steps.

1. Register the data type within Guidewire BillingCenter by creating a .dti file (data type declaration file). To do this in Studio:

- a. In the Project window, navigate to **configuration** → **config** → **datatypes**.
- b. Right-click **datatypes**, and then click **New** → **File**.
- c. Enter the name of the data type to name the file. You must add the .dti extension. Studio does not do this for you. Studio inserts this file in the correct location.
- d. The first time that you do this, you are prompted to create a new file type association for \*.dti files. In the **Register New File Type Association** dialog, click **Open matching files in Studio**, and then in the list under that option click **Text files**. Click **OK**.

You must enter definitions for the following items for the data type. If necessary, view other samples of data-type definition files to determine what you need to enter.

- Name
- Value type
- Parameters
- Implementation type

2. Create a data type definition class that implements the `gw.datatype.def.IDataTypeDef` interface. This class must include writable property definitions that correspond to each parameter that the data type accepts.
3. Create data type handler classes for each of the three aspects of the data type (constraints, persistence, and presentation). These classes must implement the following interfaces:

- `gw.datatype.handler.IDataTypeConstraintsHandler`
- `gw.datatype.handler.IDataTypePersistenceHandler`
- `gw.datatype.handler.IDataTypePresentationHandler`

Guidewire provides a number of implementations of these three interfaces for the standard data types. For example, you can create your own CLOB-based data types by defining a data type that uses the `ClobPersistenceHandler` class. To access the handler interface implementations or to view a complete list, enter the following within Gosu code:

```
gw.datatypes.impl.*
```

After you create the data type, you will want to use the data type in some useful way. For example, you can create an entity property that uses that data type and then expose that property as a field within BillingCenter.

### See also

- For a discussion of constraints, persistence, and presentation as it relates to data types, see “Overview of Data Types” on page 225.

## Defining a New Tax Identification Number Data Type

The following examples illustrates the steps involved in defining a new data type and using it. The example defines a new data type for *Tax Identification Number* objects, called `TaxID`. The data type has one required property, the name of the property on the context object. This property, `countryProperty`, identifies which country is in context for validating the data.

This example contains the following steps:

- Step 1: Register the Data Type

- Step 2: Implement the `IDataTypeDef` Interface
- Step 3: Implement the Data Type Aspect Handlers

## Step 1: Register the Data Type

To register a new data type, create a file named `xxx.dti`, with `xxx` as the name of the new data type. In this case, create a file named `TaxID.dti`. To do this:

1. In the Project window, navigate to `configuration → config → datatypes`.
2. Right-click `datatypes`, and then click `New → File`.
3. Enter `TaxID.dti` as the file name. This action creates an empty data type file and places it in the `datatypes` folder.
4. Enter the following text in the file:

```
<?xml version="1.0"?>
<DataTypeDef xmlns="http://guidewire.com/datatype" type="gw.newdatatypes.TaxIDDataTypeDef"
    valueType="java.lang.String">
    <ParameterDef name="countryProperty" desc="The name of a property on the owning entity,
        whose value contains the country with which to validate and format values."
        required="true" type="java.lang.String"/>
</DataTypeDef>
```

The root element of `TaxID.dti` is `<DataTypeDef>` and the namespace is `http://guidewire.com/datatype`.

This example defines the following:

data type name	<code>TaxID</code>
value type	<code>String</code>
parameter	<code>contactType</code>
implementation type	<code>gw.newdatatypes.TaxIDDataTypeDef</code>

### See also

- For details on the attributes and elements relevant to the data type definition, see “The BillingCenter Data Model” on page 147.

## Step 2: Implement the `IDataTypeDef` Interface

The implementation class that you create to handle the `TaxID` data type must do the following:

- It must implement the `gw.datatype.def.IDataTypeDef` interface.
- It must have a no-argument constructor.
- It must have a property for each of the data type parameters.

For example, suppose that you have a new data type that has a `String` parameter named `someParameter`. The implementation class (specified in the `type` attribute) must define a writable property named `someParameter`, so that the data type factory can pass the argument values to the implementation. The implementation can then use the parameters in the implementation of the various handlers, which are:

- `gw.datatype.handler.IDataTypeConstraintsHandler`
- `gw.datatype.handler.IDataTypePersistenceHandler`
- `gw.datatype.handler.IDataTypePresentationHandler`

### Class `TaxIDDataTypeDef`

For our example data type, the `gw.newdatatypes.TaxIDDataTypeDef` class looks similar to the following. To create this file, first create the package, then the class file, in the Studio `Classes` folder.

```
package gw.newdatatypes
uses gw.datatype.def.IDataTypeDef
```

```
uses gw.datatype.handler.IDataTypeConstraintsHandler
uses gw.datatype.handler.IDataTypePresentationHandler
uses gw.datatype.handler.IDataTypePersistenceHandler
uses gw.lang.reflect.IPropertyInfo
uses gw.datatype.handler.IDataTypeValueHandler
uses gw.datatype.def.IDataTypeDefValidationErrors
uses gw.datatype.impl.VarcharPersistenceHandler
uses gw.datatype.impl.SimpleValueHandler

class TaxIDDataTypeDef implements IDatatypeDef {
    private var _countryProperty : String as CountryProperty

    override property get ConstraintsHandler() : IDatatypeConstraintsHandler {
        return new TaxIDConstraintsHandler(CountryProperty)
    }

    override property get PersistenceHandler() : IDatatypePersistenceHandler {
        return new VarcharPersistenceHandler(/* encrypted */ false,
                                             /* trimWhitespace */ true,
                                             /* size */ 30)
    }

    override property get PresentationHandler() : IDatatypePresentationHandler {
        return new TaxIDPresentationHandler(CountryProperty)
    }

    override property get ValueHandler() : IDatatypeValueHandler {
        return new SimpleValueHandler(String)
    }

    override function validate(prop : IPropertyInfo, errors : IDatatypeDefValidationErrors) {
        // Check that the CountryProperty names an actual property on the owning type, and that
        // the type of the property is typekey.Country.
        var countryProp = prop.OwnersType.TypeInfo.getProperty(CountryProperty)

        if (countryProp == null) {
            errors.addError("Property '" + CountryProperty + "' does not exist on type " +
                           prop.OwnersType)
        } else if (not typekey.Country.Type.isAssignableFrom(countryProp.Type)) {
            errors.addError("Property " + countryProp + " does not resolve to a " + typekey.Country)
        }
    }
}
```

Note that the class defines a property named `CountryProperty`, which the system calls to pass the `countryProperty` parameter. Also notice how the implementation reads the value of `CountryProperty` as its constructs its constraints and presentation handlers. Guidewire guarantees to fill the implementation parameters before calling the handlers.

In the example code, the class refers to constraints and presentation handlers created specifically for this data type. However, it also reuses a Guidewire-provided persistence handler, the `VarcharPersistenceHandler`. You do not usually need to create your own persistence handler, as Guidewire defines persistence handlers for all the basic database column types.

### Step 3: Implement the Data Type Aspect Handlers

As you define a new data type, it is possible (actually likely) that you need to define one or more handlers for the data type. These handler interfaces are different than the Data Type API interfaces. For example, clients that use the Data Type API use the following:

```
gw.datatype.IConstrainedDataType
```

However, if you define a new data type, you must implement the following:

```
gw.datatype.handler.IDataTypeConstraintsHandler
```

This separation of interfaces allows the definition of a caller-friendly interface for data type clients and a implementation-friendly interface for data type designers.

The example data type defines a handler for both constraints and presentation.

### Class TaxIDConstraintsHandler

This class looks similar to the following:

```
package gw.newdatatypes

uses gw.datatype.handler.IStringConstraintsHandler
uses gw.lang.reflect.IPropertyInfo
uses java.lang.Iterable
uses java.lang.Integer
uses java.lang.CharSequence
uses gw.datatype.DataTypeException

class TaxIDConstraintsHandler implements IStringConstraintsHandler {

    var _countryProperty : String

    construct(countryProperty : String) {
        _countryProperty = countryProperty
    }

    override function validateValue(ctx : Object, prop : IPropertyInfo, value : Object) {
        var country = getCountry(ctx)

        switch (country) {
            case "US": validateUSTaxID(ctx, prop, value as java.lang.String)
                break
            // other countries ...
        }
    }

    override function validateUserInput(ctx : Object, prop : IPropertyInfo, strValue : String) {
        validateValue(ctx, prop, strValue)
    }

    override function getConsistencyCheckerPredicates(columnName : String) : Iterable<CharSequence> {
        return {}
    }

    override function getLoaderValidationPredicates(columnName : String) : Iterable<CharSequence> {
        return {}
    }

    override function getLength(ctx : Object, prop : IPropertyInfo) : Integer {
        var country = getCountry(ctx)

        switch (country) {
            case "US": return ctx typeis Person ? 11 : 10
            // other countries ...
        }

        return null
    }

    private function getCountry(ctx : Object) : Country {
        return ctx[_countryProperty] as Country
    }

    private function validateUSTaxID(ctx : Object, prop : IPropertyInfo, value : String) {
        var pattern = ctx typeis Person ? "\d{3}-\d{2}-\d{4}" : "\d{2}-\d{7}"
        if (not value.matches(pattern)) {
            throw new DataTypeException("${value} does not match required pattern ${pattern}", prop,
                "Validation.TaxID", { value })
        }
    }
}
```

### Class TaxIDPresentationHandler

This class looks similar to the following:

```
package gw.newdatatypes

uses gw.lang.reflect.IPropertyInfo
uses gw.datatype.handler.IStringPresentationHandler
```

```
class TaxIDPresentationHandler implements IStringPresentationHandler {  
    private var _countryProperty : String  
  
    construct(countryProperty : String) {  
        _countryProperty = countryProperty  
    }  
  
    function getEditorValue(ctx : Object, prop : IPropertyInfo) : Object {  
        return null  
    }  
  
    override function getDisplayFormat(ctx : Object, prop : IPropertyInfo ) : String {  
        return null  
    }  
  
    override function getInputMask(ctx : Object, prop : IPropertyInfo) : String {  
  
        switch (getCountry(ctx)) {  
            case "US": return ctx.type is Person ? "###-##-####" : "##-#####"  
            // other countries ...  
        }  
  
        return null  
    }  
  
    override function getPlaceholderChar(ctx : Object, prop : IPropertyInfo) : String {  
        return null  
    }  
  
    private function getCountry(ctx : Object) : Country {  
        return ctx[_countryProperty] as Country  
    }  
}
```

Notice how each of these handlers makes use of the context object in order to determine the type of input mask and validation string to use.



# Field Validation

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

This topic includes:

- “Field Validators” on page 239
- “Field Validator Definitions” on page 240
- “Modifying Field Validators” on page 243

**See also**

- “Configuring National Field Validation” on page 167 in the *Globalization Guide*

## Field Validators

Field validators handle simple validation for a single field. A validator definition defines a *regular expression*, which a data field must match to be valid. It can also define an optional *input mask* that provides a visual indication to the user of the data to enter in the field.

Each field in BillingCenter has a default validation based on its data type. For example, integer fields can contain only numbers. However, it is possible to use a field validator definition to override this default validation.

- You can apply field validators to simple data types, but not to typelists.
- You can modify field validators for existing fields, or create new validators for new fields.

For complex validation between fields, use validation-specific Gosu code instead of simple field validators.

### Specifying the Properties of a Specific Field

Field validators specify only the validation properties for a general kind of input (for example, any postal code). They do not specify the properties of a specific field in a particular data view. Instead, detail views and editable list views include additional validation attributes in their configuration files.

### Specifying Field Validators on a Delegate Entity

Apply any field validators for elements existing on a delegate entity to the delegate entity. Do not apply any field validators to the entities that inherit the elements from the delegate. Applying a field validator to an element on the delegate entity ensures that BillingCenter applies the field validator uniformly to that data element in whatever code utilizes the delegate.

#### See also

- “Field Validator Definitions” on page 240
- “Modifying Field Validators” on page 243

## Field Validator Definitions

BillingCenter stores field validator definitions in `fieldvalidators.xml` files in various locations in the `fieldvalidators` folder. These files contain a list of validator specifications for individual fields in BillingCenter. File `fieldvalidators.xml` contains the following sections:

XML element	Description
<code>&lt;FieldValidators&gt;</code>	Top XML element for the <code>fieldvalidators.xml</code> file.
<code>&lt;ValidatorDef&gt;</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 account 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 `fieldvalidators.xml` file:

```

<FieldValidators>
  <ValidatorDef name="Email"
    description="Validator.Email"
    input-mask=""
    value=".+@.+"/>
  <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>

```

In the previous example, each validator definition specifies a `value` and an `input-mask`. These attributes have different uses, as follows:

<code>value</code>	A value is a regular expression that the field value must match for the data to be valid. BillingCenter 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, assists the user in entering valid data. BillingCenter displays the input mask when the field opens for editing. For example, a # character indicates that the user must enter a digit for this character. These characters disappear when the user starts to enter data.

The input mask guides the user to enter valid sequences for the regular expression defined in the `value` attribute. After the user enters a value, BillingCenter uses the regular expression to validate the field data as it sets the field on the object.

#### See also

- “Configuring National Field Validation” on page 167 in the *Globalization Guide*

## <FieldValidators>

The <FieldValidators> element is the root element in the `fieldvalidators.xml` file. It contains the XML sub-element <ValidatorDef>.

## <ValidatorDef>

The <ValidatorDef> subelement of <FieldValidators> is the beginning element for the definition of a validator. This element has the following attributes:

- Name
- Value
- Description
- Input-Mask
- Format
- Placeholder-Char
- Floor, Ceiling

The following sections describe these attributes.

### Name

The `name` attribute specifies the name of the validator. A field definition uses this attribute to specify which validator applies to the field.

### Value

The `value` attribute specifies the acceptable values for the field. It is in the form of a regular expression. BillingCenter does not persist this value (the regular expression definition) to the database.

Use regular expressions with `String` values only. Use floor and ceiling range values for numeric fields, for example, `Money`.

BillingCenter uses the Apache library described in the following location for regular expression parsing:

<http://jakarta.apache.org/oro/api/org/apache/oro/text/regex/package-summary.html>

The following list describes some of the more useful items:

- 
- ( ) Parentheses define the order in which BillingCenter evaluates an expression, just as with any parentheses.
  - [ ] Brackets indicate acceptable values. For example:
    - [Mm] indicates the letters M or m.
    - [0-9] indicates any value from 0 to 9.
    - [0-9a-zA-Z] indicates any alphanumeric character.
  - { } Braces indicate the number of characters. For example:
    - [0-9]{5} allows five positions containing any character (number) between 0 and 9.
    - {x} repeats the preceding value x times. For example, [0-9]{3} indicates any 3-digit integer such as 031 or 909, but not 16.
    - {x,y} indicates the preceding value can repeat between x and y times. For example, [abc]{1,3} allows values such as cab, b, or aa, but not rs or abca.
  - ? A question mark indicates one or zero occurrences of the preceding value. For example, [0-9]x? allows 3x or 3 but not 3xx. ([Mm][Pp][Hh])? means mph, MpH, MPH, or nothing.
  - ( )? Values within parentheses followed by a question mark are optional. For example, (-[0-9]{4})? means that you can optionally have four more digits between 0 and 9 after a dash -.
  - \* An asterisk means zero or more of the preceding value. For example, (abc)\* means abc or abcabc but not ab.
-

- 
- + A *plus* sign means one or more of the preceding value. For example, [0-9]+ means any number of integers between 0 and 9 (but none is not an option).
  - . A *period* is a wildcard character. For example:
    - .\* means anything.
    - .+ means anything but the empty string.
    - ... means any string with three characters.
- 

## Description

The `description` attribute specifies the validation message to show to a user who enters bad input. The `description` refers to a key within the `display.properties` file that contains the actual description text. The naming convention for this display key is `Validator.validator_name`.

In the display text in the properties file, {0} represents the name of the field in question. BillingCenter determines this at runtime dynamically.

## Input-Mask

The `input-mask` attribute is optional. It specifies a visual indication of the characters that the user can enter. BillingCenter displays the input mask temporarily to the user during data entry. When the user starts to enter text, the input mask is no longer visible.

The input mask definition 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. BillingCenter displays those characters outside of the text field. For example ####mph appears as a field #### with mph on the outside end of it.

## Format

The `format` attribute works in a similar manner to the `input-mask` attribute. However, it is not currently in use.

## Placeholder-Char

The `placeholder-char` attribute specifies a replacement value for the input mask display character, which defaults to a period (.). For example, use the `placeholder-char` attribute to display a dash character instead of the default period.

## Floor, Ceiling

The `floor` and `ceiling` attributes are optional attributes that specify the minimum (`floor`) and maximum (`ceiling`) values for the field. For example, you can limit the range to 100-200 by setting `floor="100"` and `ceiling="200"`.

Use floor and ceiling range values for numeric fields only. For example, use the floor and ceiling attributes to define a Money validator:

```
<ValidatorDef description="Validator.Money"
               input-mask="" name="Money"
               ceiling="999999999999.99"
               floor="-999999999999.99"
               value=".*/>
```

## Modifying Field Validators

You configure field validation in Guidewire Studio by editing `fieldvalidators.xml` files in various locations in the `fieldvalidators` folder. In Studio, navigate in the Project window to **configuration** → **config** → `fieldvalidators`.

- You define global field validators once in the `fieldvalidators.xml` file located in the root of the `fieldvalidators` folder.
- You define national field validators in `fieldvalidators.xml` files located in country-specific packages in the `fieldvalidators` folder.

You can, for example:

- Create a new field validator. For example:

```
<!-- Create a new validator -->
<ValidatorDef name="ExampleValidator" value="[A-z]{1,5}" description="Validator.Example"
    input-mask="#####"/>
```

- Modify attributes of an existing validator. For example:

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

- Define field validators for a specific country.

### See also

- “Configuring National Field Validation” on page 167 in the *Globalization Guide*

## 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 BillingCenter interface. For example:

```
<column-override name="EmailAddressHome">
  <columnParam name="LogicalSize" value="42"/>
</column-override>
```

The use of the `logicalSize` parameter does not affect the actual length of the column in the database. It merely affects how many characters a user can enter into a text field.

# Working with Typelists

Within Guidewire BillingCenter, 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 BillingCenter that presents the set of available choices. You define and manage typelists through Guidewire Studio.

This topic includes:

- “What is a Typelist?” on page 246
- “Terms Related to Typelists” on page 246
- “Typelists and Typecodes” on page 246
- “Typelist Definition Files” on page 247
- “Different Kinds of Typelists” on page 248
- “Working with Typelists in Studio” on page 249
- “Typekey Fields” on page 252
- “Removing or Retiring a Typekey” on page 254
- “Typelist Filters” on page 255
- “Static Filters” on page 256
- “Dynamic Filters” on page 260
- “Typecode References in Gosu” on page 263
- “Mapping Typecodes to External System Codes” on page 264

**See also**

- “Localizing Typecodes” on page 50 in the *Globalization Guide*

## What is a Typelist?

**IMPORTANT** Ensure that you fully understand the dependencies between typelists and other application files before you modify a typelist. Incorrect changes to a typelist can cause damage to the BillingCenter data model.

Guidewire BillingCenter 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 BillingCenter interface, it is usually a typelist.

For example, the BillingCenter **Search Disbursements** page that you access as you edit disbursement information contains several different typelists (drop-down lists). One of these is the **Reason** typelist that provides the available values from which you can choose as you enter information about the reason for the disbursement creation.

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:

- `TroubleTicket.TicketType` uses a simple list.
- `InvoicePaymentType` uses a list with a static filter (`Deposit`).

Besides displaying the text describing the different options in a drop-down list, typelists also serve a very important role in integration. Guidewire recommends that you design your typelists so that you can map their typecodes (values) to the set of codes used in your legacy applications. This is a very important step in making sure that you code an account in BillingCenter 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 BillingCenter. 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 BillingCenter.
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 BillingCenter, 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 BillingCenter references directly. Therefore, they must exist. Studio displays internal typecodes in gray, non-editable cells. This makes it impossible for you to edit or delete an internal typecode.

### Localized Typecodes

It is possible to localize the individual typecodes in a typelist. See “Localizing Typecodes” on page 50 in the *Globalization Guide* for more information.

### Mapping Typecodes to External System Codes

See the following:

- “Mapping Typecodes to External System Codes” on page 264
- “Mapping Typecodes to External System Codes” on page 125 in the *Integration Guide*

## Typelist Definition Files

Similar to entity definitions, Guidewire PolicyCenter stores typelist definitions in XML files. There are three types of typelist files:

File type	Contains...
tti	A single typelist declaration. The name of the file prior to the extension corresponds to the name of the typelist. This can be either a Guidewire base configuration typelist or a custom typelist that you create through Studio.
txx	A single typelist extension. This can be a Guidewire-exposed base application extension or a custom typelist extension that you create.
tix	A single typelist extension for use by Guidewire only. These are generally Guidewire internal extensions to base application typelists, for use by a specific Guidewire application.

Always create, modify, and manage typelist definition files through BillingCenter Studio. Guidewire specifically does not recommend or support manipulating the XML typelist files directly.

#### See also

- “Data Entity Metadata Files” on page 149

## Different Kinds of Typelists

BillingCenter organizes typelists into the following categories:

Category	Description
Internal	<p>Typelists that Guidewire controls as BillingCenter requires these typelists for proper application operation. BillingCenter depends on these lists for internal application logic. Guidewire designates internal typelists as <i>final</i> (meaning non-extendable). Thus, Guidewire restricts your ability to modify them.</p> <p>You can, however, override the following attribute values on these types of typelists:</p> <ul style="list-style-type: none"> <li>• name</li> <li>• description</li> <li>• priority</li> <li>• retired</li> </ul>
Extendable	<p>Typelists that you can customize. These typelists come with a set of example typecodes, but it is possible to modify these typecodes and to add your own typecodes. In some cases, these extendable typelists have internal typecode values that must exist for BillingCenter to function properly. You cannot remove the internal typecodes, but you can modify any of the example typecodes.</p> <p>BillingCenter designates internal typecodes by placing their code values in gray, non-editable cells. This makes these values inaccessible, and thus, impossible to modify.</p>
Custom Typelists	<p>Typelists that you add for specific purposes, for example, to work with a new custom field. These typelists are not part of the Guidewire base configuration. Studio automatically makes all custom typelists non-final (meaning extendable).</p>

### Internal Typelists

A few of the typelists in the application are internal. Guidewire controls these typelists as BillingCenter 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 BillingCenter implements specific behavior for known activity types.

Studio indicates internal typelists by shading the typelist icon light gray in the **Resources** tree. Studio also disables your ability to add additional typecodes to internal typelists.

The following are examples of internal typelists that you cannot change:

- `ActivityType`
- `AgencyBillCycle`
- `CancellationTarget`
- `ChargePattern`
- `Incentive`

In some cases, Studio displays a typelist with a grayed-out icon in the **Resources** tree. This occurs if BillingCenter 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 BillingCenter 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. BillingCenter includes default typelists with sample typecodes in them. You can customize these typelists for your business needs by adding additional typecodes, if you want.

The **ActivityCategory** typelist is an example of an extendable typelist. If you want, you can add additional typecodes other than the sample values that Guidewire provides in the base configuration.

## Custom Typelists

If you add a new field to the application, then it is possible that you also need to add an associated typelist. You can only access these typelists through new extension fields. For more information on how to add a new field to the data model, see “Extending a Base Configuration Entity” on page 209.

To create a custom typelist, in the Project window, navigate to **configuration** → **config** → **Extensions** → **Typelist**. Right-click on **Typelist**, and then click **New** → **Typelist**. Enter a name for the typelist, and then define your typecodes. BillingCenter limits the number of characters in a typecode to 50 or less.

# Working with Typelists in Studio

You create, manage and modify typelists within BillingCenter using Guidewire Studio:

- To work with an existing extendable typelist, expand the **Typelist** folder in the Studio Project window and select the typelist from the list of existing typelists. This opens its editor in which you can change non-internal values or define new typecodes and filters.
- To view the values set for an internal typelist, select the typelist in the **Typelist** editor.
- To create a new custom typelist, navigate to **configuration** → **config** → **Extensions** → **Typelist**. Right-click on **Typelist**, and then click **New** → **Typelist**. Enter its name, and then define typecodes and filters for the typelist.

You cannot add a new typecode to, or modify an existing typecode of, a final typelist. However, it is possible to create filters for the typelist that modify its behavior within Guidewire BillingCenter.

## The Typelists Editor

If you modify an existing typelist, ensure that you thoroughly understand which other typelists depend on the typecode values in the typelist being modified. You must also update any related typelists as well. For example, any modification that you make to the **DeIniquityReason** typelist can potentially affect the **WorkflowType** typelist that the **DeIniquityReason** typelist filters. Therefore, you must update all of the related typelists as well.

After you select a typelist from the **Typelist** folder, Studio opens a typelist editor showing configuration options for that typelist.

### The Studio Typelists Editor Interface

The top portion of the **Typelist** editor contains the following fields:

- **Description**
- **Table name**
- **Final**

### The Description Field

BillingCenter transfers the value that you enter in the **New → Typelist** dialog for the type list name to the **Description** field in the typelist editor. It is possible to edit this field.

Guidewire recommends that you add a `_Ext` suffix to the value that you enter for the type list name. This ensures that the name of any typelist that you create does not conflict with a Guidewire typelist implemented in a future database upgrade.

### The Table Name Field

By default, Guidewire uses `bct1_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 `bct1_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 BillingCenter uses the **Name** value and limits the table name to a maximum of 25 characters.
- If you do provide a value for the **Table name** field, then this overrides the value that you set in the **Name** field. However, the maximum table name length is still 25 characters.

Field	Value entered in...	Maximum length	Database table name
Name	New Typelist dialog	25 characters	<code>bct1_typelist-name</code>
Table name	Typelists editor	25 characters	<code>bct1_table-name</code>

### The Final Field

A **final** typelist is a typelist to which you cannot add additional typecodes. You can, however, override the **name**, **description**, **priority**, and **retired** attributes. Studio marks typelists defined as final with a grayed-out icon. All custom typelists that you create are non-final.

## The Studio Typelists Editor Tabs

The Typelist editor screen contains a number of tabs. Some of these tabs are not visible until you make a selection in the **Codes** tab. Each tab provides different functionality.

Tab	Use to...	See...
Codes	Enter a typecode and set its attributes	• “Entering Typecodes” on page 251
Filters	Define a fixed subset of a typelist to use as a static filter.	• “Static Filters” on page 256
Categories	Create a typelist filter that depends on the typecodes in a different typelist. This is a subtab. You must select a typecode to see this tab.	• “Dynamic Filters” on page 260

### See also

- For information on how to use the Studio **Typelist Localization** editor, see “Localizing Typecodes” on page 50 in the *Globalization Guide*.

## Create a New Typelist

1. In the Project window, navigate to **configuration → config → Extensions → Typelist**.

2. Right-click on **Typelist**, and then click **New → Typelist**.
3. Enter the typelist name in the **New Typelist** dialog. BillingCenter uses this name to uniquely identify this typelist in the data model.
4. Enter a description. Use the **Description** field to create a longer text description to identify how BillingCenter uses this typelist. This text appears in places like the *Data Dictionary*.
5. Verify that the (Boolean) **Final** field is set to **false**. Studio automatically sets this field to false for any typelist that you create. You have no control over this setting. This field has the following meanings:

True	You cannot add or delete typecodes from the typelist. You can only override certain attribute fields.
False	You can modify or delete typecodes from this typelist, except for typecodes designated as internal, which you cannot delete. (You cannot remove internal typecodes, but you can modify their name, description, and other fields.)

## Entering Typecodes

You use the **Codes** tab to enter typecodes for this typelist and to set various attributes for the typecodes. Each typecode represents one value in the drop-down list. Every typelist must have at least one typecode. Within this tab, you can set the following:

Field	Description
Code	A unique ID for internal Guidewire use. Enter a string containing only letters, digits, or the following characters: <ul style="list-style-type: none"><li>• a dot (.)</li><li>• a colon (:)</li></ul> Do not include white space or use a hyphen (-). Use this code to map to your legacy systems for import and export of BillingCenter data. The code must be unique within the list. BillingCenter limits the number of characters in a typecode to 50 or less. See also “Mapping Typecodes to External System Codes” on page 264.
Name	The text that is visible within BillingCenter 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. BillingCenter displays the text in this field in the <i>BillingCenter Data Dictionary</i> .
Priority	A value that determines the sort order of the typecodes (lowest priority first, by default). You use this to sort the codes within the drop-down list and to sort a list of activities, for example, by priority. If you omit this value, BillingCenter sorts the list alphabetically by name. If desired, you can specify priorities for some typecodes but not others. This causes BillingCenter 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. BillingCenter does not make changes to any existing objects that reference this typecode. If you do not enter a value, BillingCenter assumes the value is <b>false</b> (the default value).

### Naming New Typecodes

Guidewire recommends that you add a **\_Ext** suffix to the **Code** value for any new typecodes that you create. Do this only if the **Code** value is legal on any external system that needs to use the value. If that value is not legal, then omit the **\_Ext** suffix.

### Maximum Typelist Size

Guidewire strongly recommends that you limit the maximum number of typecodes in a typelist to 255 items. Any number larger than that can cause performance issues. If you need more than 255 typecodes, 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 8192 typecodes on a typelist.

### Typelists and the Data Model

Guidewire recommends that you regenerate the *Data Dictionary* after you add or modify a typelist. Guidewire does not require that you do this. However, regenerating the *Data Dictionary* is an excellent way to identify any flaws with your new or modified typelist.

During application start up, Guidewire upgrades the application database if there are any changes to the data model, which includes any changes to a typelist or typecode. (In actual practice, this only occurs if the `autoupgrade` option is set to `true` in `config.xml`, which is almost always the case.)

#### See also

- “Typelists and Typecodes” on page 246
- “Mapping Typecodes to External System Codes” on page 264
- “Mapping Typecodes to External System Codes” on page 125 in the *Integration Guide*

## Typekey Fields

A *typekey field* is an entity field that BillingCenter 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 BillingCenter field to use a typelist to set values requires the following:

1. The typelist must exist. If it does not exist, then you must create it using the **Typelist** editor in BillingCenter 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 BillingCenter.

### 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 BillingCenter Studio, through the **Typelists** editor. For information on using the **Typelists** editor, see “Working with Typelists in Studio” on page 249.

## 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" />
```

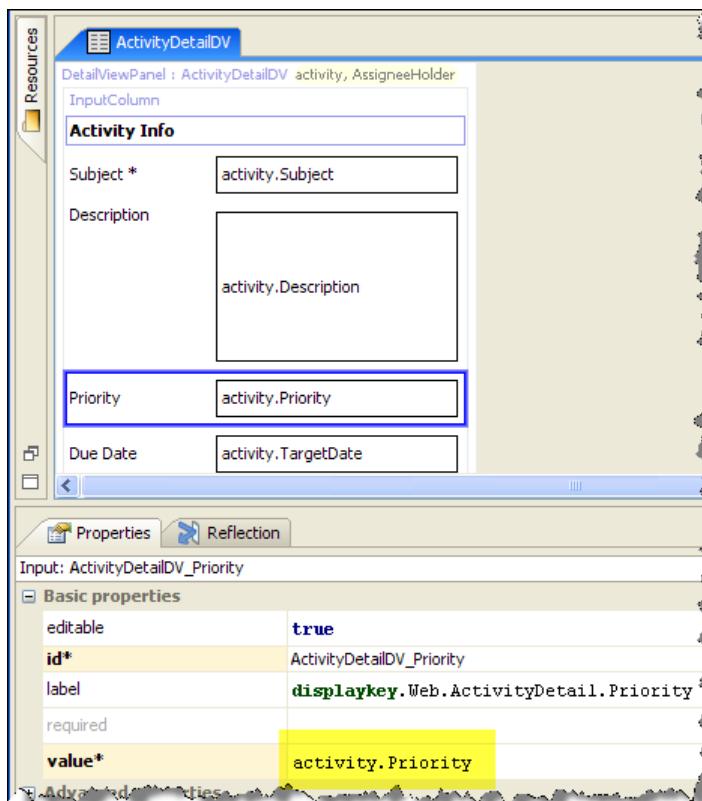
### See also

- For information on the `<typekey>` element, see “`<typekey>`” on page 194.
- For information on how to create data model entities, see “The BillingCenter Data Model” on page 147.
- For information on how to modify existing data model entities, see “Modifying the Base Data Model” on page 205.

## Step 3: Reference the Typelist in the PCF File

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

Thus, the BillingCenter **ActivityDetailDV** 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 269.
- For information on working with PCF files in general, see “Introduction to Page Configuration” on page 283.

#### Step 4: Update the Data Model

Guidewire recommends that you regenerate the *BillingCenter 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 BillingCenter interface. Restarting the application server forces BillingCenter to upgrade the data model in the application database.

## Removing or Retiring a Typekey

Ensure that you fully understand the dependencies between typelists and other application files before you modify a typelist.

In general, Guidewire does not recommend that you make changes to existing typelists other than the following:

- Extending a non-final typelist to add additional typekeys.
- Retiring a typekey, which makes it invisible in the BillingCenter interface, but leaves the typekey in the application database.

Be very careful of removing typekeys from a typelist as it is possible that multiple application files reference that particular typekey. Removing a typekey incorrectly can cause the application server to not start. Guidewire recommends that you retire a typekey rather than remove it.

It is not possible to remove a typekey from a typelist marked as final. It is also not possible to remove a typekey marked as internal. BillingCenter indicates internal typekeys by placing their typecode values in gray, non-editable cells. This makes these typecode values inaccessible, and thus, impossible to modify.

## Removing a Typekey

Suppose that you delete the `email_sent` typekey from the base configuration `DocumentType` typelist for some reason. If you remove this typekey, then you must also update all others part of the application install and disallow the production of documents of that type. In particular, you must remove references to the typekey from any `.descriptor` file that references that typekey. In this case, a search of the document template files finds that the `CreateEmailSent.gosu.htm.descriptor` file references `email_sent`.

### To remove a typekey

1. Navigate to the typelist that contains the typekey that you want to retire.
2. Click the typekey, and then click Remove .
3. Search for additional references to the typekey in the application files and remove any that apply. Pay particular attention to `.descriptor` files. To remove a typekey reference:
  - a. Perform a case-insensitive text search throughout the application files to find all references to the deleted typekey.
  - b. Open these files in Studio and modify as necessary.

### To retire a typekey

1. Navigate to the typelist that contains the typekey that you want to retire.
2. Select the **Retired** cell of the typekey that you want to retire.
3. Set the cell value to **true**.

If you retire a typekey, Guidewire recommends that you perform the steps outlined in *To remove a typekey* to identify any issues with the retirement:

- Verify all Studio resources.
- Perform a case-insensitive search in the application files for the retired typekey.

## Typelist Filters

It is possible to configure a typelist so that BillingCenter filters the typelist values so that they do not all appear in the drop-down list (typelist) in the BillingCenter 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"><li>• Include certain specific typecodes on the typelist only.</li><li>• Include certain specific categories of typecodes on the typelist.</li><li>• Exclude certain specific typecodes from the full list of the typecodes on the typelist</li></ul>	"Static Filters" on page 256
Dynamic	A dynamic subset of the values on a typelist. You can create filters that: <ul style="list-style-type: none"><li>• Associate one or more typecodes on a parent typelist with one or more typecodes on a child typelist.</li><li>• Associate all the typecodes on a parent typelist with one or more typecodes on a child typelist.</li></ul>	"Dynamic Filters" on page 260

## Static Filters

A *static* typelist filter causes the typelist to display only a subset of the typecodes for that typelist. Therefore, a static filter narrows the list of typecodes to show in the typelist view in the application. Guidewire calls this kind of typelist filter a static *typefilter*.

You define a static filter at the level of the typelist. You do this through the Studio Typelists editor, by defining a filter on the **Filters** tab for that particular typelist.

Studio manages the typelist XML file for you automatically. If you examine this file, you see that Studio uses the following XML syntax to define a static typelist filter. (In this case, a static filter that defines—or includes—a subset of the available typecodes.)

```
<typelistextension xmlns="http://guidewire.com/typelists" desc="Yes, no or unknown" name="YesNo">
  <typecode code="No" desc="No" name="No" priority="2"/>
  <typecode code="Yes" desc="Yes" name="Yes" priority="1"/>
  <typecode code="Unknown" desc="Unknown" name="Unknown" priority="3"/>
  <typefilter desc="Only display Yes and No typelist values" name="YesNoOnly">
    <include code="Yes"/>
    <include code="No"/>
  </typefilter>
</typelistextension>
```

Notice that the XML declares each typecode on the typelist (Yes, No, and Unknown). It then specifies a filter named **YesNoOnly** that limits the available values to simply Yes and No. This is static (fixed) filter.

For more information on the **<typefilter>** element, see “**<typekey>**” on page 194.

### To create a static filter

1. Define the typecodes for this typelist in the Studio Typelist editor. See “Working with Typelists in Studio” on page 249 for details.
2. Select the **Filters** tab on this typelist in the **Typelist** editor.
3. Click **Add** and enter the following information for your static filter:

Attribute	Description
Name	The name of the filter. BillingCenter uses this value to determine if a field uses this filter.
Description	Description of the context for which to use this typefilter.
Include All?	(Boolean) Typically, you only set this value to <b>true</b> if you use the <b>exclude</b> functionality. <ul style="list-style-type: none"> <li>• <b>True</b> indicates that the typelist view starts with the full list of typecodes. You then use exclusions to narrow down the list.</li> <li>• <b>False</b> (the default) instructs BillingCenter to use values set in the various subpanes to modify the typelist view in the application.</li> </ul>

4. Use the fields in the following panes on the **Filters** tab to create a fixed subset of the typecodes for use in the static filter.

Subpane	Use to...	See...
Categories	Specify one or more typecodes to include by category within the filtered typelist view.	“Creating a Static Filter Using Categories” on page 257
Includes	Specify one or more typecodes to include within the filtered typelist view.	“Creating a Static Filter Using Includes” on page 258
Excludes	Specify one or more typecodes to exclude from the full list of typecodes for this typelist.	“Creating a Static Filter Using Excludes” on page 259

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 252 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 BillingCenter application bin directory to regenerate the *Data Dictionary*:

```
gwbc regen-dictionary
```

7. Stop and restart the application server to update the data model.

## Creating a Static Filter Using Categories

Suppose that you want to filter a list of United States cities by state. (Say that you want to only show a list of appropriate cities if you select a certain state.) To create this filter, you need to first to define a City typelist (if one does not exist). You then need to populate the typelist with a few sample cities:

City typecodes	Location
ABQ	Albuquerque, NM
ALB	Albany, NY
LA	Los Angeles, NM
NY	New York, NY
SF	San Francisco, CA
SND	San Diego, CA
SNF	Santa Fe, NM

Then, for each City typecode, you need to set a category, similar to the following. You do this by selecting each typecode in turn, then clicking **Add** in the **Categories** pane in the **Codes** tab and entering the appropriate information:

City typecode	Associated typelist	Associated typecode
ABQ	State	NM
ALB	State	NY
LA	State	CA
NY	State	NY
SF	State	CA
SND	State	CA
SNF	State	NM

To generalize this example to regions outside the United States, you could associate the **Jurisdiction** typelist and a specific jurisdiction with each city typecode instead.

After making your choices, you have something that looks similar to the following:

Code	Name	Description	Priority	Retired
ABQ	Albuquerque	Albuquerque, NM	-1	false
SNF	Santa Fe	Santa Fe, NM	-1	false
LA	Los Angeles	Los Angeles, CA	-1	false
SND	San Diego	San Diego, CA	-1	false
NY	New York	New York, NY	-1	false
ALB	Albany	Albany, NY	-1	false

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 BillingCenter user interface.
- Reference the typelist in the PCF file in which you want to display the typelist.

See “Typekey Fields” on page 252 for more information on declaring a typelist on an entity and referencing that typelist in a PCF file. In general, though, you need to add something similar to the entity definition that want to display the typelist:

```
<typekey name="NewMexico" typelist="City" typefilter="NewMexico" nullok="true"/>
```

## Creating a Static Filter Using Includes

Suppose that you want to create a filtered typelist that displays zone codes that are in use only in Canada and not any other country. One way to create the filter is to use an **Includes** filter on the **ZoneTypes** typelist.

In this example, you want the typelist to display only the following:

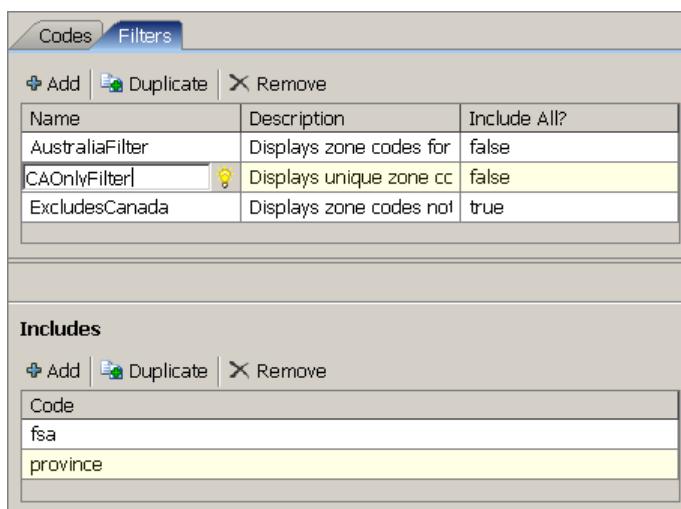
- fsa
- province

The following table shows the typecodes in the ZoneTypes typelist:

ZoneType typecode	Associated typelist	Associated typecode	Include in filter
city	Country	CA (Canada) US (United States)	
county	Country	US	
fsa	Country	CA	●
locality	Country	AU (Australia)	
postcode	Country	AU	
province	Country	CA	●
state	Country	AU US	
zip	Country	US	

#### To create an Include filter

1. Open the typelist that you want to filter in the Studio Typelists editor.
2. Navigate to the Filters tab.
3. Add the filter name to the list of filters. For example, call the filter that only displays certain zone type for the country of Canada CAOnlyFilter.
4. Finally, add the typecodes you want to include in the typelist in the Includes pane.



#### Creating a Static Filter Using Excludes

Suppose 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 in the ZoneTypes typelist except for the following:

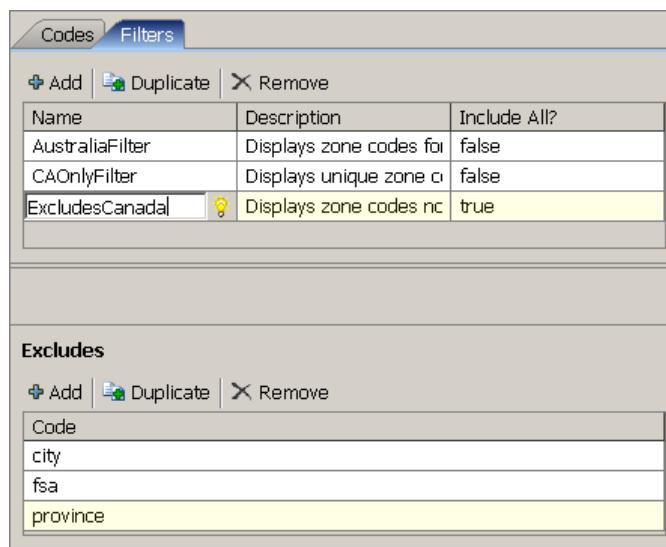
- city
- fsa
- province

The following table shows the typecodes in the ZoneTypes typelist:

ZoneType typecode	Associated typelist	Associated typecode	Include in filter
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 use a parent typecode to restrict the available values on the child typecode.

You define a typecode filter directly on a typecode. You do this through the Studio Typelist editor, by defining a filter on the Codes tab for a particular typecode. To create this filter, you select a specific typecode and set a filter, or *category*, on that typecode.

There are two types of typecode filters that you can define on the **Codes** tab:

Filter type	Use to...
Category	Associate one or more typecodes on a parent typelist with one or more typecodes on a child typelist.
Category list	Associate all the typecodes on a parent typelist with one or more typecodes on a child typelist.

### Category Typecode Filters

- You use a category filter to associate one or more typecodes from one or more typelists with a specific typecode on the filtered typelist.
- You define a category filter in the **Typelist** editor on the **Codes** tab using the **Categories** pane.

Studio manages the typelist XML file for you automatically. If you examine this file, you see that Studio uses the following XML syntax to define a typecode category filter:

```
<typecode code="DependentTypecode" desc="DescriptionString" typelist="DependentTypelistName">
  <category code="Typecode1" typelist="Typelist1"/>
  <category code="Typecode2" typelist="Typelist1"/>
  <category code="Typecode3" typelist="Typelist2"/>
...
</typecode>
```

### Category List Typecode Filters

- You use a category list filter to associate all of the typecodes from one or more typelists with a specific typecode on the filtered typelist.
- You define a category list filter in the **Typelists** editor on the **Codes** tab using the **Category Lists** pane.

Studio manages the typelist XML file for you automatically. If you examine this file, you see that Studio uses the following XML syntax to define a typecode category list filter:

```
<typecode code="Typecode" desc="DescriptionString" typelist="DependentTypelistName">
  <categorylist typelist="TypelistName"/>
</typecode>
```

## Creating a Dynamic Filter

In general, to create a dynamic filter, you need to do the following:

- Step 1: Set the Category Filter on Each Typecode
- Step 2: Declare the Category Filter on an Entity
- Step 3: Set the BillingCenter 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 BillingCenter, a user with administrative privileges can define a new activity pattern (**Administration** → **Activity Patterns** → **New Activity Pattern**). Within the **New Activity Pattern** screen, you see several drop-down lists:

- Type
- Category

BillingCenter 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:

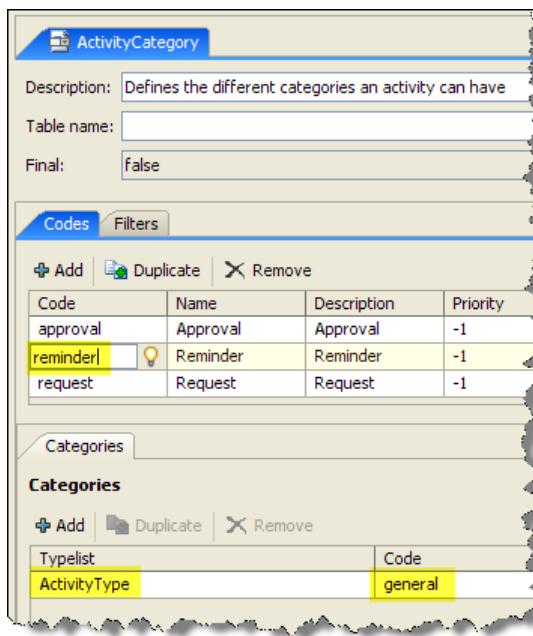
- Reminder
- Request

The **ActivityCategory** typelist is the typelist that controls what you see in the **Category** field in BillingCenter. 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, BillingCenter filters each individual typecode in this typelist so that it is only available for selection if you first select the associated typelist and typecode.

### Step 1: Set the Category Filter on Each Typecode

The process is the same to create a category list typecode filter. In that case, you associate a single typelist (and all its typecodes) with each individual typecode on the dependent typelist. You make the association by selecting a typecode in the dependent typelist and setting the controlling typelist in the **Category Lists** pane.

Open the **ActivityCategory** typelist and select each typecode in turn. As you do so, you see that Studio associates each typecode with an **ActivityType.Code** value in the **Categories** pane. For example if you select the **reminder** 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 BillingCenter **Category** field? The following code sample illustrates what you need to do. You must add a typekey for both the parent (**ActivityType**) typelist and the dependent child (**ActivityCategory**) typelist.

```

<entity xmlns="http://guidewire.com/datamodel" ... entity="ActivityPattern" ...>
  ...
  <typekey default="general" desc="Type of the activity." name="Type" typelist="ActivityType"/>
  ...
  <typekey ... name="Category" typelist="ActivityCategory">
    <keyfilters>
      <keyfilter name="Type"/>
    </keyfilters>
  </typekey>
  ...
</entity>

```

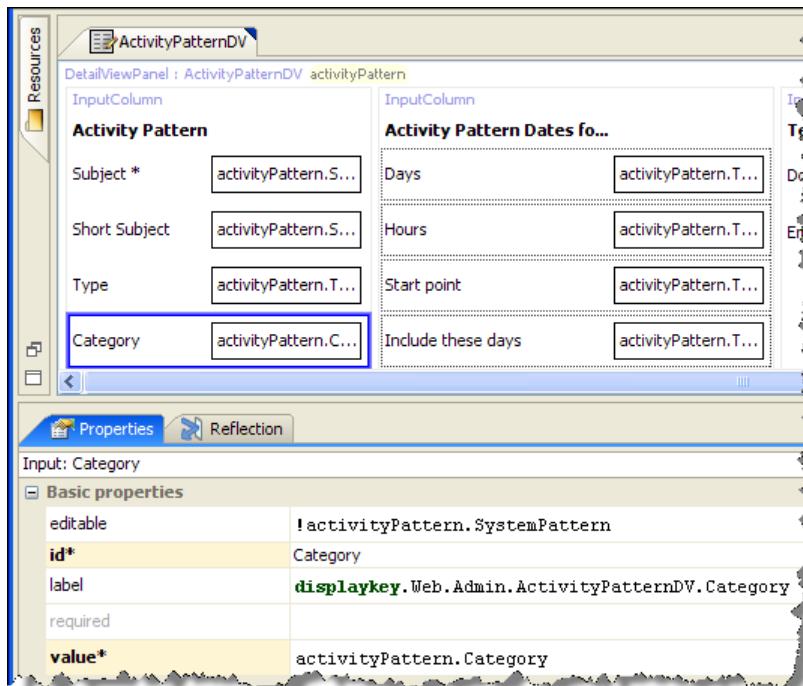
The sample code first defines a <typekey> element with name="Type" and typelist="ActivityType". This is the controlling (parent) typelist. The code then defines a second typelist (ActivityCategory) with a keyfilter name="Type". It is the typelist referenced by the <keyfilter> element that controls the behavior of the typelist named in the <typekey> element. Thus, the value of ActivityType.Code controls the associated typecode on the dependent ActivityCategory typelist.

For more information on the <keyfilter> element, see “<typekey>” on page 194.

### Step 3: Set the BillingCenter 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 BillingCenter New Activity Pattern screen. To access an entity typelist, you need to use the entity.TypeList syntax. For example, to access the ActivityCategory typelist on the ActivityPattern entity, use ActivityPattern.Category with Category being the name of the typelist.

You do this in the BillingCenter ActivityPatternDV.pcf file:



### Step 4: Update the Product Model

Guidewire recommends that you regenerate the *BillingCenter 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 BillingCenter interface. Restarting the application server forces BillingCenter to upgrade the data model in the application database.

## Typecode References in Gosu

To refer to a specific typecode in Gosu, use the following syntax.

`typekey.TypeList.TC_Typecode`

For example, the default State typelist has typecodes for states in the US and provinces in Canada.

State	
Code	Name
IL	Illinois
...	...

To refer to the typecode for the state of Illinois in the typelist State, use the following Gosu expression.

typekey.STATE.TC\_IL

You must prefix the code in the object path expressions for typecodes with TC\_.

**Note:** Use code completion in Studio to build complete object path expressions for typecodes. Type “typekey.” to begin, and work your way down to the typecode that you want.

## Mapping Typecodes to External System Codes

Your BillingCenter application can share or exchange data with one or more external applications. If you use this functionality, Guidewire recommends that you configure the BillingCenter 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 BillingCenter typecode, but the external applications do not match. Alternatively, suppose that you extend your typecode schema in BillingCenter. This can possibly cause a situation in which three different codes in BillingCenter represent a single (less granular) code in the other application.

To handle these more complex cases, you need to edit resource file `typecodemapping.xml` within Guidewire Studio. (You can find this file in the `configuration → config → typelists.mapping` folder.) This file specifies a namespace for each external application. Then, you identify the individual unique typecode maps by typelist.

### A Typecode Mapping Example

The following code sample illustrates a simple `typecodemapping.xml` file:

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

  <typelist name="AccountSegment">
    <mapping typecode="PR" namespace="accounting" alias="ACT" />
  </typelist>
</typecodemapping>
```

The `namespacelist` tag contains one or more `namespace` tags—one for each external application. Then, to map the actual codes, you specify one or more `typelist` tags as required. Each `typelist` tag refers to a single internal or external typelist in the application. The `typelist`, in turn, contains one or more `mapping` tags. Each `mapping` tag must contain the following attributes:

---

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

---

In the previous example, the PR BillingCenter code maps to an external application named `accounting`. You can create multiple mapping entries for the same BillingCenter typecode or the same name space. For example, the following specifies a mapping between multiple BillingCenter codes and a single external code:

```
<typelist name="BoatType">
  <mapping typecode="AI" namespace="accounting" alias="boat" />
  <mapping typecode="HY" namespace="accounting" alias="boat" />
</typelist>
```

After you define the mappings, from an external system you can use the `TypeListToolsAPI` web service to translate the mappings. See the “Mapping Typecodes to External System Codes” on page 125 in the *Integration Guide*.



# User Interface Configuration



# Using the PCF Editor

This topic covers how to work with PCF (Page Configuration Format) files in Guidewire Studio.

This topic includes:

- “Page Configuration (PCF) Editor” on page 269
- “Page Canvas Overview” on page 270
- “Creating a New PCF File” on page 270
- “Working with Shared or Included Files” on page 271
- “Page Config Menu” on page 273
- “Toolbox Tab” on page 274
- “Structure Tab” on page 274
- “Properties Tab” on page 275
- “PCF Elements” on page 277
- “Working with Elements” on page 277

## Page Configuration (PCF) Editor

Guidewire BillingCenter uses *page configuration format* (PCF) files to render the BillingCenter interface. You use the PCF editor in Studio to manage existing PCF files and create new ones.

The PCF editor provides the following features:

- Intelligent Gosu coding
- Instant feedback whenever a PCF file changes
- Drag-and-drop composition of PCF pages and their graphical elements
- High-level view of PCF page groupings
- Ability to localize the display keys used in a PCF page

The PCF editor comprises three areas:

- In the center pane, the graphical page *canvas*, which provides drag-and-drop capabilities for composing and managing the graphical and interactive elements on a page.
- In the right-hand pane, the following tabs:
  - **Toolbox** – Contains a search box and a list of elements that you can insert into the page
  - **Structure** – Shows the containment hierarchical of the elements on the page
- At the bottom, the **Properties** tabs at the bottom of the screen.

## Page Canvas Overview

The center pane of the PCF editor provides the graphical page *canvas*. The page canvas provides drag-and-drop capabilities for composing and managing the graphical and interactive elements on a page.

The page canvas displays the following:

- Elements that represent page content, such inputs and similar items, in simplified versions to illustrate how they appear within the BillingCenter user interface.
- Elements that function primarily as containers (data views, for example) as light gray boxes, with a header indicating the element type and ID.
- Elements that define or expose additional Gosu symbols to their descendants as light gray boxes, with a list of symbols at the top. If you move your mouse over a symbol, Studio shows a tooltip with the name, type, and initial value of the symbol.
  - If the symbol represents a Require, the tooltip indicates this as well.
  - If you click a symbol name, Studio selects the containing element, and then opens the appropriate properties tab for editing whatever is providing the symbol. Finally, if necessary, Studio selects the symbol in the **Properties** tab.
- Elements that are conditionally visible with a dotted border.
- Elements that iterate over a set of data and produce their contents once for each element in the data by a single copy of the contents. It follows this with an ellipsis to indicate iteration.
- RowIterator widgets with inferred header and footer cells in the position in which they appear within BillingCenter.

## Creating a New PCF File

Guidewire Studio displays PCF files in an organizational hierarchy. To create a new PCF file, you need to first decide its location in the PCF hierarchy. If the hierarchy does not contain a PCF folder at the organization level that suits your needs, first create one before you create your new PCF File.

PCF folder names are case-insensitive and must be unique within the PDF hierarchy. You cannot create a PCF folder name that differs from an existing PCF folder name by case only.

### To create a new PCF folder

1. In the Project window, navigate to **configuration** → **config** → **Page Configuration**, and expand it.
2. Select a node one level above the level in which you need to create the new PCF folder (node).
3. Right-click and click **New** → **PCF folder**.
4. Enter the folder name in the **New Folder** dialog.

### To create a new PCF file

1. In the Project window, navigate to **configuration** → **config** → **Page Configuration**, and expand it.

2. Select the node in which you want to create the new PCF file.
3. Right-click and click New → PCF file.
4. Enter the file name in the New PCF File dialog.
5. Select the PCF file type to create.
6. Enter a mode. (Any element that dynamically includes this widget must specify the same mode.) This field is only active with specific file types. See “Working with Shared or Included Files” on page 271 for more information.

The following table lists the file type icons.

Icon	File type	Icon	File type	Icon	File type	Icon	File type
	Page		Input Set		Navigation Tree		Toolbar Buttons
	Popup		List View		Panel Row		Wizard
	Card View		List-Detail View		Panel Set		Wizard Steps
	Chart View		Location Group		Popup Wizard		Wizard Step Subgroup
	Detail View		Menu Actions Set		Row Set		Worksheet
	Entry Point		Menu Items		Screen		
	Exit Point		Menu Links Set		Tab Bar		
	Info Bar		Navigation Forward		Template Page		

## Working with Shared or Included Files

A *shared element* or *shared section* is any PCF element that has the following characteristics:

- The PCF element is not a top-level element, meaning it is not a Page, Popup, or Wizard, for example.
- The PCF element exists in its own file.

Guidewire calls this a shared section because it is possible to share the element (or file) between multiple top-level elements. BillingCenter automatically propagates any changes that you make to the shared section to all other PCF elements that include the shared section.

You cannot select elements within the included file and the included elements do not display a highlight or a tooltip as you move the mouse cursor over it. For all intents and purposes, included elements are flat content of the element in the current file that includes them.

However, BillingCenter displays a PCF element that includes the contents of another file or element with a blue overlay. This overlay is cumulative. Studio displays included elements that are several levels deep in a darker shade of blue. If you double-click an area with a blue overlay, Studio opens the included file in a new editor view.

Right-clicking anywhere on the canvas and toggling **Show included sections** or toggling **Show included sections** from the **Page Config** menu disables the representation of the included files. Studio displays the text of the reference expression instead.

## Understanding PCF Modes

Certain included files or elements are *modal*. The basic idea is that you can define several different file versions, or modes, of a single shared section. Thus, any PCF page that includes the section can decide at run-time which file version to use, possibly based on the value of some variable.

If it is possible for the PCF file to have multiple modal versions, then you see **Shared section mode** above the shared area (which Studio shades or high-lights in blue). If you click the mode, Studio shows a drop-down of all the possible modes. You can use the drop-down to select a different modal file. If you do so, then Studio updates the screen to reflect your change.

For example, in ClaimCenter (the other Guidewire applications provide similar examples), PCF file `ExposureDetailScreen` contains a shared area. The mode drop-down contains a number of possible modal files that you can embed into the `ExposureDetailsScreen`. In this screen, the drop-down shows the following:

- Baggage
- Bodilyinjurydamage
- Content
- EmployerLiability
- ...

To determine if a PCF file has multiple modal versions, you can also look at the PCF file names in Studio. If you see multiple file names that include a common name followed by a dot then a different name, then this is a modal file. For example, in ClaimCenter, you see the following under the `exposures` node in the Resources tree:

- `ExposureDetailDV.Baggage`
- `ExposureDetailDV.Bodilyinjurydamage`
- `ExposureDetailDV.Content`
- `ExposureDetailDV.Employerliability`
- ...

Each individual file is a modal version of the `ExposureDetailDV` PCF file, which you can embed into another file, in this case, the `ExposureDetailsScreen`.

## Setting a PCF Mode

It is only possible to set a *mode* on a PCF file as you create that PCF file. Selecting a file type that allows modes enables the **Mode** text field in the **New PCF File** dialog. Selecting a file type that does not allow modes disables the **Mode** text field.

You are able to set a mode with the following file types only:

- |                            |                   |                        |
|----------------------------|-------------------|------------------------|
| • Accelerated Menu Actions | • List View       | • Row Set              |
| • Card View                | • Menu Action Set | • Screen               |
| • Chart View               | • Menu Items      | • Toolbar Buttons      |
| • Detail View              | • Menu Links Set  | • Wizard Steps         |
| • Info Bar                 | • Modal Cell      | • Wizard Step Subgroup |
| • Input Set                | • Panel Set       |                        |

Typically, you use a Gosu expression to define the mode for an included section. You can make this expression either a hard-coded string literal or you can use the expression to evaluate a variable or a method call. For example, PCF file `AddressPanelSet` (in PolicyCenter) uses `selectedAddress.CountryCode` as the mode expression variable.

A hard-coded string guarantees that the included section always uses the same mode regardless of the data on the page. If using a hard-coded string expression, Studio shows only that mode and does not show a drop-down above the blue area.

## Creating New Modal PCF files

It is not possible to change or modify the mode of a base configuration PCF file. You can, however, use an existing modal file as a template to create a new (different) modal version of that file. To do this:

- Select the template file and duplicate it.
- Select the newly created file and change the mode of that file.

For example, suppose that you wanted to add a new modal version of the `ExposureDetailDV` PCF file, say `ExposureDetailDV.BusinessPropertydamage`. To do this:

1. Select the file that you intend to use as the template. For this example, select `ExposureDetailDV.Baggage`.
2. Right-click the template file and select **Duplicate**.
3. Enter the name of the new modal file in the **Duplicate PCF File** dialog and click **OK**. For this example, enter `ExposureDetailDV.BusinessPropertydamage`  
Studio inserts the new file into the directory structure, colors the file name blue, and opens a view of the file automatically.
4. Modify the new modal file as required.

### Include Files with Multiple Modes.

BillingCenter provides the ability to use a single include file with multiple modes. In this way, you can re-use a single modal file in multiple PCF files. For example, in the base configuration, ClaimCenter defines PCF file `AddressBookAdditionalInfoInputSet.PersonVendor` with multiple modes:

- `PersonVendor`
- `Attorney`
- `Doctor`

To see this, open `AddressBookAdditionalInfoInputSet.PersonVendor` and select the entire file. (You see a solid blue line surrounding the file.) Examine the `mode` attribute in the **Properties** pane at the bottom of the screen. You see the following:

`PersonVendor|Attorney|Doctor`

### To create an include file with multiple modes

1. Select the include file.
2. Right-click and select **Change mode....**
3. Enter the individual modes separated by a pipe symbol ( | ) in the **Change Mode** dialog.

## Page Config Menu

If you open the PCF editor, Studio displays a **Page Config** menu on the main Studio menu bar. This menu contains a number of useful items.

Menu command	Use to...	See
<code>Change element type...</code>	Substitute a different element for the selected element. The dialog contains a list of element types that you can substitute for the selected element within the constraints of the PCF schema.	" <a href="#">Changing the Type of an Element</a> " on page 279
<code>Edit comment...</code>	Attach a comment to any element on the canvas.	" <a href="#">Adding a Comment to an Element</a> " on page 279
<code>Delete comment</code>	Remove a comment from an element.	" <a href="#">Adding a Comment to an Element</a> " on page 279

Menu command	Use to...	See
Disable element	Disable an element by commenting out the widget. This prevents BillingCenter from rendering the widget in the interface.	"Adding a Comment to an Element" on page 279
Enable element	Enable a previously disabled element. This action removes the surrounding comment tags from the element.	"Adding a Comment to an Element" on page 279
Link widgets	Link widgets on a parent page that spans multiple child PCF files. You use this particularly for explicit iterator references.	"Linking Widgets" on page 281
Show included sections	Toggle the visibility of child files embedded in a parent PCF file. If you disable the representation of the included files, Studio displays the text of the reference expression instead.	"Page Canvas Overview" on page 270
Find by ID	Find an element by its ID. The dialog contains a filter text field and a list of all elements on the canvas that have their id attribute set:	"Finding an Element on the Canvas" on page 280
Show element source	View the XML code for an element. Studio displays the XML code in a pop-up window.	"Viewing the Source of an Element" on page 280

## Toolbox Tab

The **Toolbox** tab contains a search box and a list of widgets, divided into categories and subcategories.

- Clicking on a category name expands or collapses that category.
- Clicking on a subcategory name expands or collapses that subcategory as well.

Within the toolbox, Studio persists the state of each category (expanded or collapsed) across all PCF editor views. It also persists the state of each category to each new Studio session.

Studio only displays widget categories containing widgets that are valid and available for use in the current PCF file. If you hover the mouse cursor over a widget name in the list, then Studio displays a description of that widget in a tooltip.

### Search Box

You use the search box to filter the full set of widgets. Typing in the search box temporarily expands all widget categories and highlights:

- Any widgets whose category name matches the typed text
- Any widgets whose name matches the typed text
- Any widgets whose actual name in the XML matches the typed text
- Any widgets whose description contains the typed text

Clicking the X icon by the search box clears text from the box and stops filtering the widget list. Keyboard shortcut ALT+/ gives focus to the search box.

## Structure Tab

The **Structure** tab shows the hierarchical structure of the PCF file as a tree. Each node in the tree represents a PCF element. Any children of the node are children of that element:

- If you click an element that represents a concrete element on the canvas, Studio selects that element on the canvas.

- If you click on an element that does not represent a concrete element on the canvas, then Studio first selects the containing element on the canvas. It then selects the appropriate properties tab with which to edit the clicked element. Finally, if necessary, Studio selects the clicked element in the properties tab (at the bottom of the screen).

## Properties Tab

The **Properties** tab (at the bottom of the screen) displays all attributes of the selected element. Studio divides the attribute workspace into **Basic** and **Advanced** sections. You can expand or collapse a workspace section by clicking the title of that section. Studio maintains the expanded or collapsed state of a section across all element selections and persists this state to new Studio sessions.

The workspace displays each attribute as a row in a table, with the attribute name in the left column and the value in the right column. Studio grays out the name if you have not set a value for that attribute (if the attribute value is nothing or is only a default value). Studio also grays out the attribute value if it is a default value. If the PCF schema requires an attribute, Studio displays the attribute name in bold font, with an asterisk, and with a different background color.

If you hover the mouse cursor over an attribute name, Studio displays a tooltip with the documentation for that attribute.

For each attribute:

- If the attribute takes a non-Gosu string value, Studio displays the value in a text field.
- If the attribute takes a non-Gosu Boolean value, Studio displays the value in a drop-down menu with two choices, `true` and `false`.
- If the attribute takes an enumeration value, Studio displays the value in a drop-down menu with a choice for each value of the enumeration, plus a `<none selected>` option.
- If the attribute takes a Gosu value, Studio displays the value in a single-line Gosu editor. Gosu editor commands that operate on multiple lines have no effect in a single-line editor. (For example, the `SmartFix Add uses statement` command does not work in a single-line editor.) Studio displays the single-line editor with a red background if it contains any errors, and a yellow background if it contains warnings but no errors.
- If the attribute requires a return type, Studio colors the value background red under the following circumstances:
  - If the entered expression does not evaluate to that type
  - If the entered statement does not return a value of that type
- If the attribute requires a Boolean return value, Studio displays a drop-down menu on the right side with two options, `true` and `false`. If you select one of these options, Studio sets the text of the editor to the appropriate value.

If the value editor for an attribute has focus, Studio displays the attribute name in a different background color and adds an X icon. If you click the X icon, Studio sets the value of the attribute to its default.

If you press `Enter` on the keyboard while editing a property, Studio moves the focus to the next property in the list.

## Child Lists

Some of the Properties tabs contain a *child list*. A child list contains a list of the selected element's child elements of a certain type, and a properties list for the selected child element. You can perform a number of operations on a child list, using the following tool icons.

- If the child list represents a single child type, Studio adds a new child element. Studio selects the newly added child automatically. If the child list represents multiple child types, Studio opens a drop-down menu of available child types. If you select a child type from the drop-down, Studio adds a child of that type and selects it automatically.
- If you select a child and click the delete icon, Studio removes the selected child. Studio disables this action if there is no selected child.
- If you select a child and click the up icon, Studio moves the selected child above the previous child. Studio disables the up icon if you do not first select a child, or if there are no other children above your selected child.
- If you click the down icon, Studio moves the selected child below the next child. Studio disables the down icon if you do not first select a child, or if there are not other children below your selected child.

## Additional Properties Tabs

Depending on the children of a selected element, Studio displays additional subtabs.

Additional tabs	Description
Axes	If an element can have DomainAxis and RangeAxis children, Studio displays the Axes properties tab. This tab contains a child list of the DomainAxis and RangeAxis children for the selected element. If you select a RangeAxis, Studio displays a child list for its Interval children.
Code	If an element can have a Code child, Studio displays the Code properties tab. This tab contains a Gosu editor for editing the contents of the Code child. The Code editor has access to all the top-level symbols in the PCF file (for example, any required variables). However, you cannot incorporate any uses statements, nor does the Gosu editor provide the SmartFix to add uses statements automatically.
Data Series	If an element can have DataSeries and DualAxisDataSeries children, Studio displays the Data Series properties tab. This tab contains a child list of the DataSeries and DualAxisDataSeries children for the selected element.
Entry Points	If an element can have LocationEntryPoint children, Studio displays the Entry Points properties tab. This tab contains a child list of the LocationEntryPoint children for the selected element.
Exposes	If a parent PCF page contains an iterator that controls a ListView element defined in a separate PCF file, then Studio displays the Exposes tab on the child PCF file. You use this tab to set the iterator to use for the ListView element.
Filter Options	If an element can have ToolbarFilterOption and ToolbarFilterOptionGroup children, Studio displays the Filter Options properties tab. This tab contains a child list of the ToolbarFilterOption and ToolbarFilterOptionGroup children for the selected element.
Next Conditions	If an element can have NextCondition children, Studio displays the Next Conditions properties tab. This tab contains a child list of the NextCondition children for the selected element.
Reflection	If an element can have a Reflect child, Studio displays a Reflection properties tab. The Reflection tab has a checkbox for Enable client reflection, which indicates whether that element has a Reflect child. If you enable client reflection, the Reflection tab also contains a properties list for the Reflect element and a child list for its ReflectCondition children.
Required Variables	If an elements can have Require children, Studio displays the Required Variables properties tab. This tab contains a child list of the Require children for the selected element.
Scope	If an element can have Scope children, Studio displays the Scope properties tab. This tab contains a child list of the Scope children for the selected element.
Sorting	If an element can have IteratorSort children, Studio displays the Sorting properties tab. This tab contains a child list of the IteratorSort elements for the selected element.

Additional tabs	Description
Toolbar Flags	If an element can have ToolbarFlag children, Studio displays the Toolbar Flags properties tab. This tab contains a child list of the ToolbarFlag children of the selected element.
Variables	If an element can have Variable children, Studio displays the Variables properties tab. This tab contains a child list of the Variable children for the selected element.

## PCF Elements

Studio displays a down arrow icon to the right of a non-menu element that contains menu-item children.

- If you click the down arrow, Studio opens a pop-up containing the children of the element.
- If you click anywhere on the canvas outside the pop-up, Studio dismisses the pop-up.

Studio displays elements that contain a comment with a comment icon in the upper right-hand corner of the widget. It shows disabled elements (commented-out elements) in a faded-out manner

Studio displays elements that cause a verification error with either a red overlay or a thick red border. It displays elements that cause a verification warning (but not an error) with either a yellow overlay or a thick yellow border.

If you move your mouse over an element:

- Studio highlights the element with a light border.
- If the element has a comment, Studio displays the text of the comment in a tooltip.
- If the element does not have a comment, but does have its desc attribute set, Studio displays the value of the desc attribute in a tooltip.
- If the element has any errors or warnings, Studio displays these in a tooltip along with any comment or desc text.

## PCF Elements and the Properties Tab

If you click an element on the canvas, Studio selects that element and highlights it in a thick border. This action also opens the Properties tab in the workspace area at the bottom of the screen, if it is not already visible.

- If the element has an error or warning that is attributable to one of its attributes, Studio highlights that attribute in the Properties tab.
- If the element contains child elements not shown on the canvas, Studio displays additional Properties tabs in the workspace area.
- If the element has no errors or warnings, but a non-visible child element does, Studio brings the appropriate Properties tab for that child element to the front. If necessary, Studio selects that child element in the Properties tab.
- If there are additional Properties tabs that do not apply to the selected element, Studio closes them.
- If the tab that was at the front before you selected the element is still visible, it remains at the front. Otherwise, Studio brings the Properties tab to the front.

Clicking in the canvas area outside the area representing the file being edited, or clicking Escape, de-selects the currently selected element and closes all open Properties tabs.

## Working with Elements

Page configuration format files contain three basic types of elements:

- Physical elements (buttons and inputs, and similar items) that have a visual presence in a live application.
- Behavioral elements (iterator sorting and client reflection, and similar items) that exist only to specify behavior of other elements.

- Structural elements (panels, screens, and similar items) that do not represent a single element in the Web interface, but instead indicate some grouping or other structure.

After you create a new page, you can select page elements from the **Toolbox** tab for inclusion in the page.

BillingCenter does not permit you to insert elements that are invalid for that page or grouping. After adding an element to a page, you can change its type if needed, rather than removing it and starting again.

Guidewire strongly recommends that you label the widgets that you create with unique IDs. Otherwise, you may find it difficult to identify that widget later.

You can perform the following actions with PCF elements:

- Adding an Element to the Canvas
- Changing the Type of an Element
- Adding a Comment to an Element
- Finding an Element on the Canvas
- Viewing the Source of an Element
- Duplicating an Element
- Deleting an Element
- Copying an Element
- Cutting an Element
- Pasting an Element

## Adding an Element to the Canvas

To add a widget, click its name in the **Toolbox** and hold the mouse cursor down. As you begin to drag the widget, Studio changes the mouse cursor so that it includes the icon for that widget. Studio places a green line on the canvas at every location on the canvas that it is possible to place the widget. Studio highlights the green line that is nearest on the canvas to the cursor. Studio also overlays in green the element containing the highlighted green line.

- If the widget is a menu item of some sort, Studio overlays in green those widgets that can accept the item as a menu item.
- If a green widget is closer to the cursor than any of the green lines, Studio overlays it with a brighter green.

If you click Esc, Studio cancels the action, returns the cursor to normal, and makes the green lines and overlays disappear.

If you click again (or end the dragging operation), Studio adds the new widget at the location of the highlighted green line. (Or, Studio adds the widget as a child of the highlighted widget.) Studio sets all attributes of the new widget to their default value.

After Studio adds the new widget to the canvas, the cursor returns to normal and the green lines and overlays disappear. Studio selects this new widget automatically.

## Moving an Element on the Canvas

If you click on a widget on the canvas, Studio picks up (selects) the widget. As you drag the widget, Studio moves the widget from its current location to the new location. This makes no changes to the attributes or descendants of the widget.

You can also CTRL+drag a widget on the canvas. This time, however, as you place the widget, Studio creates a duplicate of the original widget (including all attributes and descendants) and places the cloned widget at the target location.

## Changing the Type of an Element

If you right-click an element and select **Change element type**, Studio opens the **Change Element Type** dialog. You can also select the element and then select **Change element type** from the **Page Config** commands on the menu bar.

This dialog contains a list of element types that you can substitute for the selected element within the constraints of the PCF schema. If you then select a new element type and click **OK**, Studio replaces the selected element with an element of the new type. It also transfers all attribute values and descendants that are valid on the new type.

However:

- If it is possible to select a new element type that does not allow one or more attributes supported by the selected (existing) element. In this case, Studio displays a message that indicates which attributes it plans to discard.
- If it is possible to select an element type that can not contain one or more children of the selected widget. In this case, Studio displays a message that indicates which children it plans to discard.

If there are no valid element types to which you can change the selected element, Studio disables the **Change element type** command.

## Adding a Comment to an Element

It is possible to attach a comment to any element on the canvas. Studio indicates an element has a comment by placing a yellow note icon in the comment's upper right corner. If you hover the mouse over that element, then Studio displays the comment in a tooltip.

### Adding a Comment

If you do one of the following, Studio opens a modal dialog with a text field for the element's comment:

- Right-click an element and select **Edit comment**
- Select the element and then select **Edit comment** from the **Page Config** commands on the menu bar

If the element already has a comment, Studio pre-populates the text field with the contents of the comment.

### Deleting a Comment

If you do one of the following, Studio deletes the comment for that element:

- Right-click an element and select **Delete comment**
- Select the element and then select **Delete comment** from the **Page Config** commands on the menu bar

However, if the element has no comment, Studio disables the **Delete comment** command.

### Disabling (Commenting-out) an Element

Commenting out a widget effectively prevents BillingCenter 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 BillingCenter from rendering the widget. In this case, however, the XML file retains the widget. It still exists server-side at run time, although BillingCenter does not render it, and the widget does not post data. Thus, commenting out the widget can possibly give you a marginal performance increase. Also, disabling the widget prevents it from causing errors, for example, if the signature of some function called by one of its attributes changes.

As it is the use of XML comment tags that disable the widget, you cannot then add a comment to the widget to describe why you disabled it. If you would like to add an explanation associated with the widget (recommended), then use the `widget desc` attribute. Studio displays this text in the tooltip if you hover the mouse over the widget in the PCF editor. (This does not produce a yellow note icon, however.)

### Enabling an Element

If you do one of the following, Studio enables the element by removing the surrounding comment tags:

- Right-click a disabled element and select **Enable element**.
- Select the element and click **CTRL+/.**
- Select **Enable element** from the **Page Config** commands on the menu bar.

### Finding an Element on the Canvas

If you do one of the following, Studio opens a semi-modal dialog. This dialog contains a filter text field and a list of all elements on the canvas that have their `id` attribute set:

- Right-click in the canvas area and select **Find by ID**.
- Click **CTRL+F12**.
- Select **Find by ID** from the **Page Config** commands on the menu bar.

As you type in the text field, Studio filters the visible elements to those whose ID matches the typed text. Selecting an element from the list selects it on the canvas.

### Viewing the Source of an Element

To view the XML representation of an element, select the widget, then do one of the following:

- Right-click, and then select **Show element source**.
- Select **Show element source** from the **Page Config** menu.

Studio opens a small text window and displays the XML code associated with the selected element.

### Duplicating an Element

If you do one of the following, Studio creates a duplicate of the element immediately after the current element:

- Right-click an element and select **Duplicate**.
- Select a widget and click **CTRL+D**.
- Select **Duplicate** from the **Edit** commands on the menu bar.

This includes all attribute values and descendants. Studio selects the duplicate widget automatically.

If the PCF schema permits the target widget to occur one time only within the parent widget (for example, a Screen), then attempting to duplicate the widget has no effect.

### Deleting an Element

If you do one of the following, Studio deletes the element from the canvas:

- Right-click an element and select **Delete**.
- Select a widget and click **Delete**.
- Select **Delete** from the **Edit** commands on the menu bar.
- Select the **Delete** icon from the menu bar.

You cannot delete the root element of a PCF.

## Copying an Element

If you do one of the following, Studio copies an XML representation of that widget and its descendants to the clipboard:

- Right-click an element and select **Copy**.
- Select a widget and click CTRL+C.
- Select **Copy** from the **Edit** commands on the menu bar.
- Select the **Copy** icon from the menu bar.

## Cutting an Element

If you do one of the following, Studio copies an XML representation of that widget and its descendants to the clipboard and deletes the widget:

- Right-click an element and select **Cut**.
- Select a widget and click CTRL+C.
- Select **Cut** from the **Edit** commands on the menu bar.
- Select the **Cut** icon from the menu bar.

You cannot cut (remove) the root element of a PCF file.

## Pasting an Element

If the content of the clipboard is valid XML representing a PCF widget, you can paste the widget by doing one of the following:

- Right-click the canvas and select **Paste**.
- Click CTRL+V.
- Select **Paste** from the **Edit** commands on the menu bar.
- Select the **Paste** icon from the menu bar.

## Linking Widgets

A common feature in PCF pages is a **ListView** element controlled by a **RowEditor** iterator. BillingCenter renders the list view as a table with multiple rows, with the iterator populating the data in the table rows. Frequently, the user clicks a button to activate certain functionality within the list view, for example, adding or deleting rows in the table.

In many cases, the PCF file is a parent page that contains embedded child PCF files that contain individual **ListView** elements. If this is the case, then you need to link the button on the parent PCF file with the iterator used to populate the child PCF files. To do so, you use the **Link widgets** command on the **Page Config** menu. This menu item provides a visual tool to link widgets on a parent page that spans multiple child PCF files. You use this particularly for explicit iterator references.

### To link two widgets

1. Select a widget, for example a **CheckedValuesToolbarButton** widget.
2. Do one of the following:
  - Select **Link widgets** from the **Page Config** menu.
  - Right-click and select **Link widgets** from the context menu.
  - Press CTRL+L.

Studio changes the look of the mouse cursor to cross-hairs. Studio also changes the color of the target widget to light green.

3. Click the widget to which you want to link. Studio links the two widgets.

# Introduction to Page Configuration

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

This topic includes:

- “Page Configuration Files” on page 283
- “Page Configuration Elements” on page 283
- “Getting Started Configuring Pages” on page 288
- “Modifying Style and Theme Elements” on page 290

## Page Configuration Files

The pages in the BillingCenter user interface are defined by XML files stored within each installed instance of the application. To configure your BillingCenter 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*.

PCF files are stored in `BillingCenter/modules/configuration/config/web/pcf`. Guidewire does not support editing PCF files outside of Guidewire Studio.

## Page Configuration Elements

This section discusses the following topics:

- What is a PCF Element?
- Types of PCF Elements
- Identifying PCF Elements in the User Interface

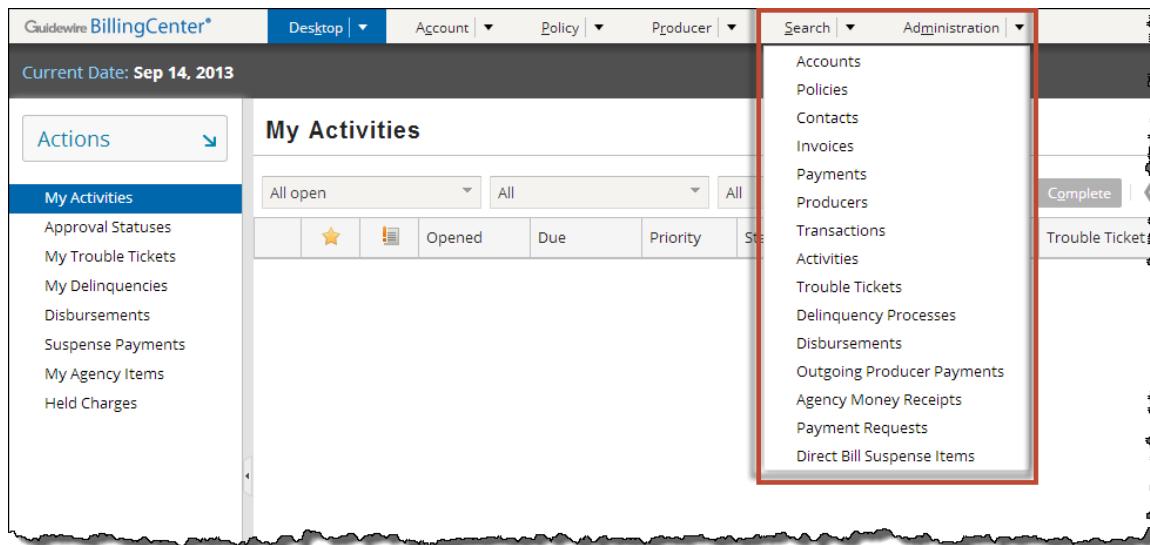
## What is a PCF Element?

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

- Editors
- List views
- Detail views
- Buttons
- Popups
- Other BillingCenter interface elements
- Non-visible objects that support the BillingCenter interface elements, such as Gosu code that performs background actions after you click a button.

For a reference of all PCF elements and their attributes, see the *PCF Format Reference* in `BillingCenter/modules/pcf.html` in your installation.

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

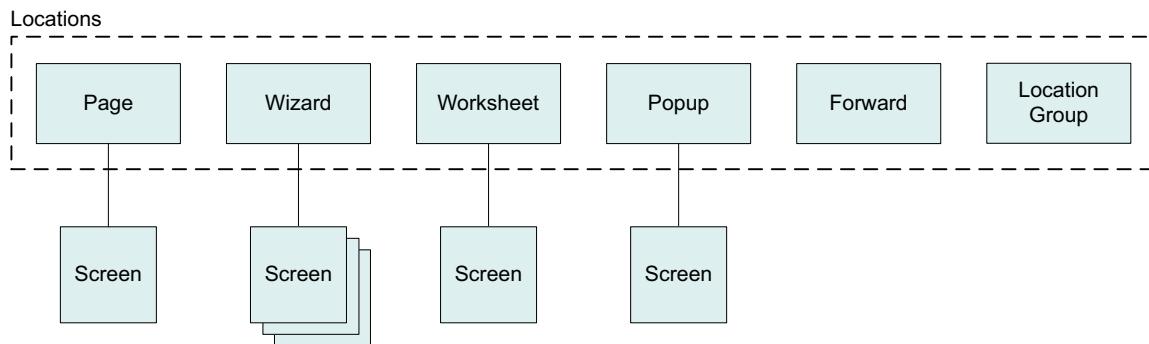
There are many kinds of PCF elements that you can define. These elements follow a hierarchical, container-based user interface model. To design them most effectively, you need understand the relationships between them thoroughly. Most PCF elements are of one of the following types:

- Locations
- Widgets

## Locations

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

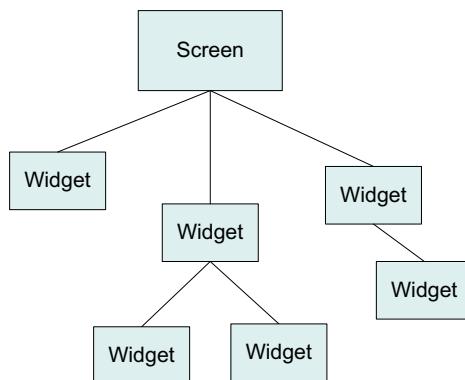
Location	Description
Page	A location with exactly one screen. The majority of locations defined in BillingCenter are pages.
Wizard	A location with one or more screens, in which only one screen is active at a time. The contents of a wizard are usually not defined in PCF files, but are configured either in other configuration files or are defined internally by BillingCenter.
Worksheet	A page that can be shown in the workspace, the bottom pane of the web interface. The main advantage of worksheets is that they can be viewed at the same time as regular pages. This makes them appropriate for certain kinds of detail pages such as creating a new note.
Popup	A page that appears on top of another page, and that returns a value to its invoking page. Popups allow users to perform an interim action without leaving the context of the original task. For example, a page that requires the user to specify a contact person could provide a popup to search for the contact. After the popup closes, BillingCenter 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. BillingCenter 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 BillingCenter can render into HTML. BillingCenter 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 BillingCenter 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 BillingCenter, you must first understand how it is constructed. This includes understanding the PCF elements which compose the page, what files define the PCF elements, and how they are pulled together.

For example, consider the Claim Summary page within ClaimCenter. If you look at this page in the ClaimCenter interface, you cannot immediately tell how it is constructed. If you want to modify this page, some of the important things to know about it are illustrated in the following annotated diagram:

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

**Panel set: *ClaimSummaryHeadlinePanelSet***

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

**Detail view: *ClaimSummaryDV***

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

**List view: *ServiceRequestLV***

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

This diagram shows:

- The location is a page named *ClaimSummary*.
- The page contains a screen named *ClaimSummaryScreen*.
- The screen contains a “panel set” widget named *ClaimSummaryHeadlinePanelSet*.

- The screen contains a “detail view” widget named `ClaimSummaryDV`.
- The screen contains a “list view” widget named `ServiceRequestLV`.

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

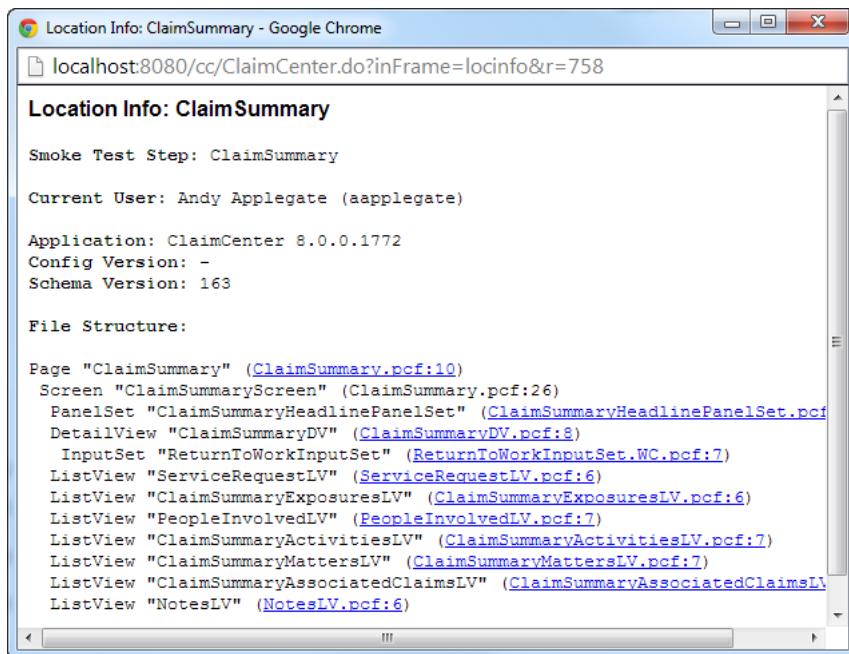
- Location Info
- Widget Inspector

To enable these tools, the `EnableInternalDebugTools` configuration parameter must be set to `true`.

## Location Info

The **Location Info** window shows you information about the construction of the page you are viewing. It includes the location name, screen names, and high-level widgets defined in the page, and the names of the PCF files in which they are all defined. Typically, the widgets that appear in this window are the ones that are defined in separate files, such as screens, detail views, list views, and so on. The **Location Info** is most useful if you are making changes to a page as it tells you which files you need to modify.

To view the location information for a particular page, go to that page in the BillingCenter interface, and then press **ALT+SHIFT+I**. This pops up the **Location Info** window for the active page. For example, the following is the **Location Info** window for the ClaimCenter **Claim Summary** page:



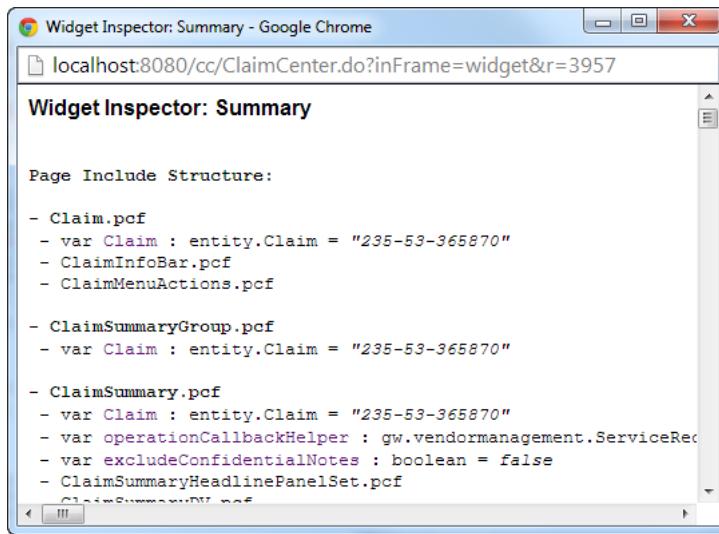
With this information, you can see:

- The location is a page named `ClaimSummary`, defined in the `ClaimSummary.pcf` file on line 10.
- The page contains a screen named `ClaimSummaryScreen`, defined in the `ClaimSummary.pcf` file on line 26.
- The screen contains one detail view widget, and multiple list view widgets, each defined in a different file.

## Widget Inspector

The **Widget Inspector** shows detailed information about the widgets that appear on a page. This includes the widget name, ID, label text, and the file in which it is defined. The widget information is most useful during debugging a problem with a page. For example, suppose that a defined widget does not appear on a page. You could then look at the widget information to determine whether the widget exists (but perhaps is not visible) or does not exist at all.

To view the widget inspector for a particular page, go to that page in the BillingCenter 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

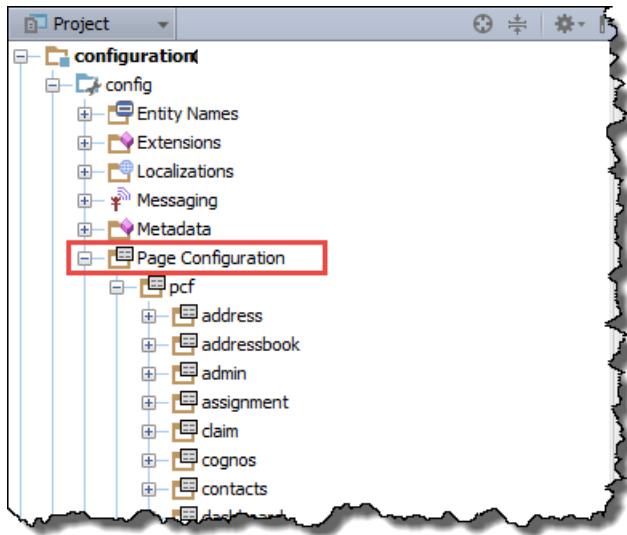
### Finding an Existing Element To Edit

The first step in modifying the BillingCenter 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

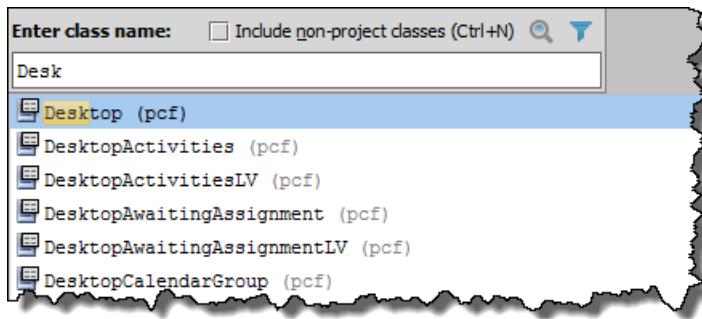
You can browse the PCF elements under the **Page Configuration** folder in Guidewire Studio:



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



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

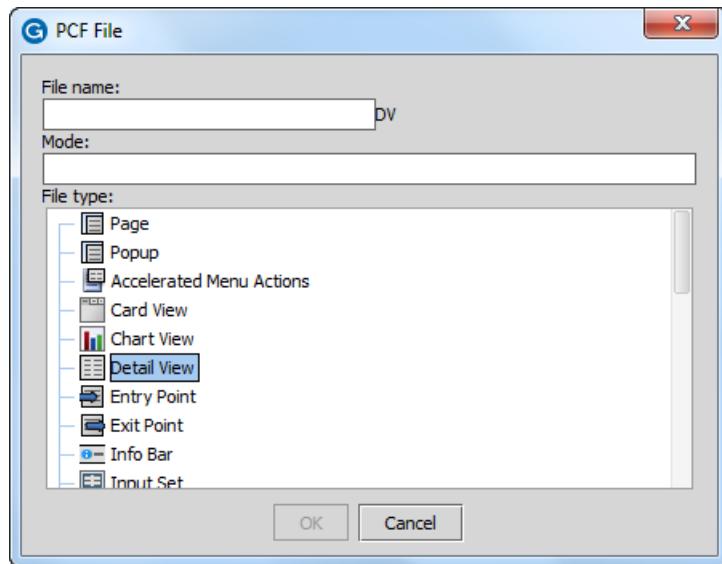
## Creating a New Standalone PCF Element

You can create a new PCF standalone element in Guidewire Studio. Each standalone element is stored in its own file.

### To create a new standalone element

1. Browse the **Page Configuration** folder in the **Project** window, and locate the folder under which you want to create your new element.

2. Right-click on that folder, and then click **New → PCF File**. (You can also click **New → PCF Folder** to create a new folder). The **PCF File** dialog appears.



3. In the **File name** text box, type the name of the element.
4. Click the type of element to create. If an element has a naming convention, it is shown next to the **File name** text box. For example, the name of a detail view must end with **DV**.
5. Click **OK**, and the new element is created and opened for editing in Studio.

## Modifying Style and Theme Elements

### Changing or Adding Images

Images used in the application reside in `BillingCenter/modules/configuration/webresources/themes/Titanium/resources/images`. Images can be switched by replacing an existing image file with one of the same name. We recommend ensuring that replaced images are the same size as the original to avoid sizing issues.

To add a new image, place it in this folder or one of its children, then reference it as appropriate.

To update your application with these new images, run `gwbc update-theme` from the command line.

### Overriding CSS

To override specific CSS classes, make edits to `BillingCenter/modules/configuration/webresources/themes/Titanium/resources/theme_ext.css`. Changes in this file override other CSS properties in your application.

### Changing Theme Colors

Guidewire applications are themed using SASS technology. To make significant changes to the style of your application, see “Advanced Re-Theming” on page 291. However, you can change the theme colors of your application by following these steps:

1. Open `BillingCenter/ThemeApp/packages/titanium/sass/var/Component.scss` in a text editor. This is where all of the BillingCenter colors are defined.

2. You can modify the definition to be any other hexadecimal color, or to be relative to another. For example, `$base-light-color` takes the `$base-color` and lightens it appropriately across the application.
3. After making changes, run `gwbc update-theme` from the command line. This incorporates your changes into the CSS generated by the SASS Theme.

For more information about SASS, visit <http://sass-lang.com>.

## Advanced Re-Theming

BillingCenter uses SASS to define a robust set of styling rules for ensuring a consistent look across the application. To make more detailed styling and theme changes, we recommend referencing the SASS documentation for details. There are, however, a few things specific to the Guidewire implementation:

- You cannot create a new theme and apply it to the application. All changes to the styling need to be made in the current theme definition, under `BillingCenter/ThemeApp/packages/titanium/sass/`.
- SASS condenses its CSS definition for performance purposes. To see the expanded, debuggable CSS in your web development tool, run the command `gwbc dev-deploy-web-resources-debug` and refresh your browser.
- `gwbc update-theme` re-condenses your CSS for production use.



# Data Panels

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

This topic includes:

- “Panel Overview” on page 293
- “Detail View Panel” on page 293
- “List View Panel” on page 298

## Panel Overview

A *panel* is a widget that contains the visual layout of the data to display in a screen. There are several types of panels:

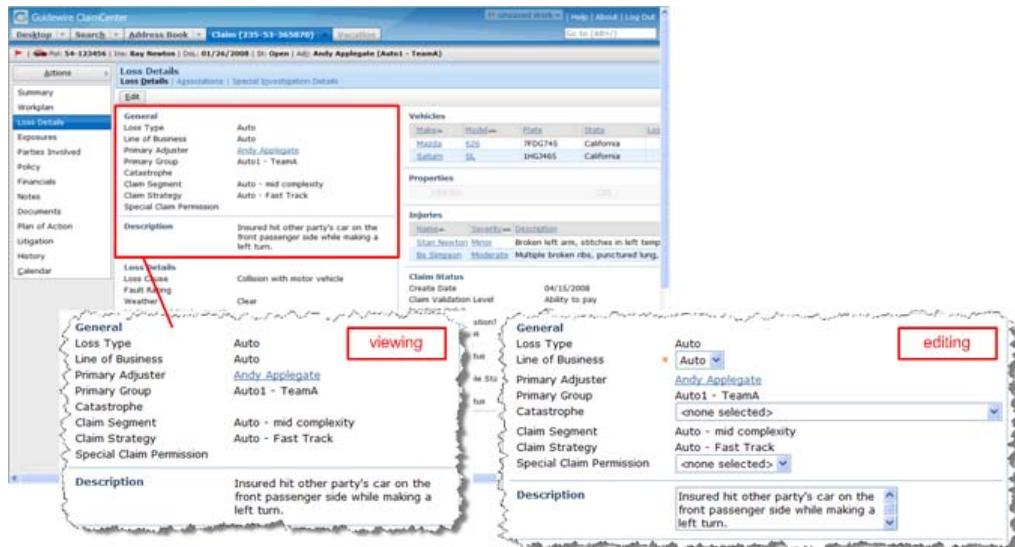
- Detail View Panel – A series of widgets laid out in one or more columns.
- List View Panel – A list of array objects, or any other data that can be laid out in tabular form.

You can place as many panels in a screen as you like, dividing the screen into one or more areas.

## Detail View Panel

A *detail view* is a panel that is composed of a series of data fields laid out in one or more columns. It can contain information about a single data object, or it can include data from multiple related objects. Any input widget can appear within a detail view.

The following is an example of a detail view as it appears both as it is being viewed and as it is being edited:

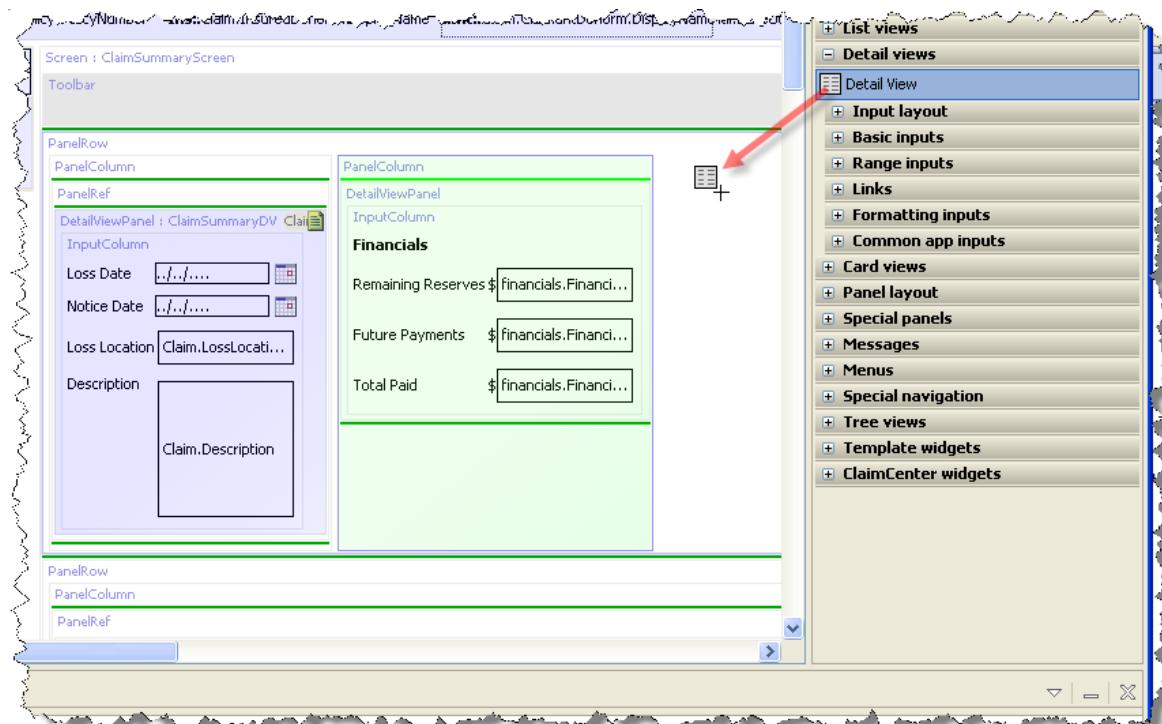


You can do the following:

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

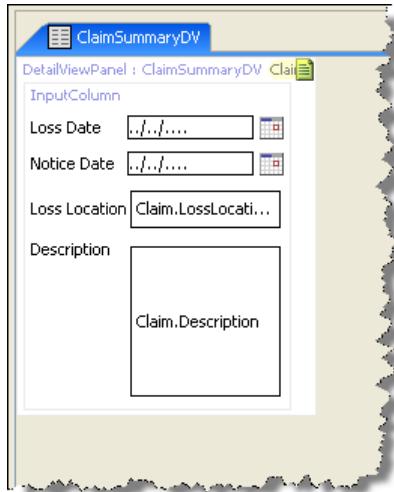
## Define a Detail View

Define a detail view by dragging the Detail View element onto the PCF canvas. You can place the element anywhere a green line appears. For example:

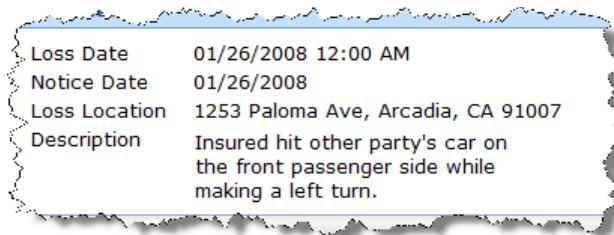


The `id` attribute is required; it identifies the panel so that it can be referenced by other PCF elements. The ID must be unique, and it must end with the text string DV.

A detail view must contain at least one vertical column, defined by the `Input Column` element. The column contains the input widgets to display, as in the following example:



This definition produces the following detail view:



## Add Columns to a Detail View

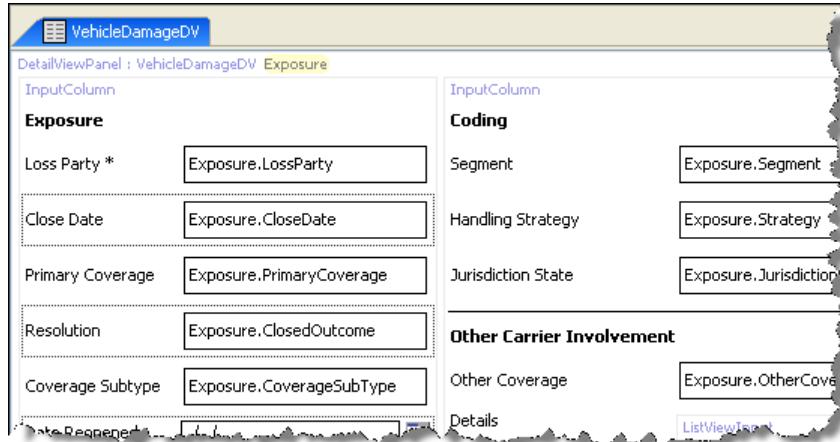
A detail view must contain at least one vertical column, but it can contain more. The following illustration shows detail views with one and two columns:

Column 1

Column 1

Column 2

A column is defined by the `Input Column` element. This element must appear at least once, to define the first column. To add additional columns, include the `Input Column` element multiple times. The following example defines a two-column detail view:



BillingCenter automatically places a vertical divider between the columns.

The full definition of the previous example produces the following two-column detail view:

<b>Exposure</b>		<b>Coding</b>	
Loss Party	Insured's loss	Segment	Auto - low complexity
Primary Coverage	Liability - Property damage	Handling Strategy	Auto - Fast Track
Coverage Subtype	Liability - Property Damage - Vehicle	Jurisdiction State	California
Coverage			
Adjuster	<a href="#">Andy Applegate</a>		
Group	Auto1 - TeamA		
Status	Open		
Create Date	04/15/2008		
Statistical Line	-		
Validation Level			
<b>Claimant</b>		<b>Financials</b>	
Claimant	<a href="#">Ray Newton</a>	Remaining Reserves	-
Type	Owner of other vehicle	Future Payments	-
		Total Paid	-
		Total Recoveries	-
		Net Total Incurred	-

vertical divider

## Format a Detail View

You can add the following formatting options to a detail view:

- Label
- Input Divider

These are illustrated in the following diagram:

The diagram shows a detail view panel with the following structure:

<b>General</b>	
Loss Type	Auto
Line of Business	Auto
Primary Adjuster	<a href="#">Andy Applegate</a>
Primary Group	Auto1 - TeamA
Catastrophe	
Claim Segment	Auto - mid complexity
Claim Strategy	Auto - Fast Track
Special Claim Permission	
<b>Description</b>	
Insured hit other party's car on the front passenger side while making a left turn.	
<b>Loss Details</b>	
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:

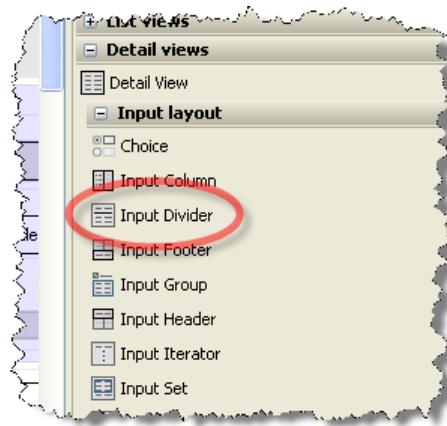


Set the `label` attribute to the display key to use for the label.

## 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.

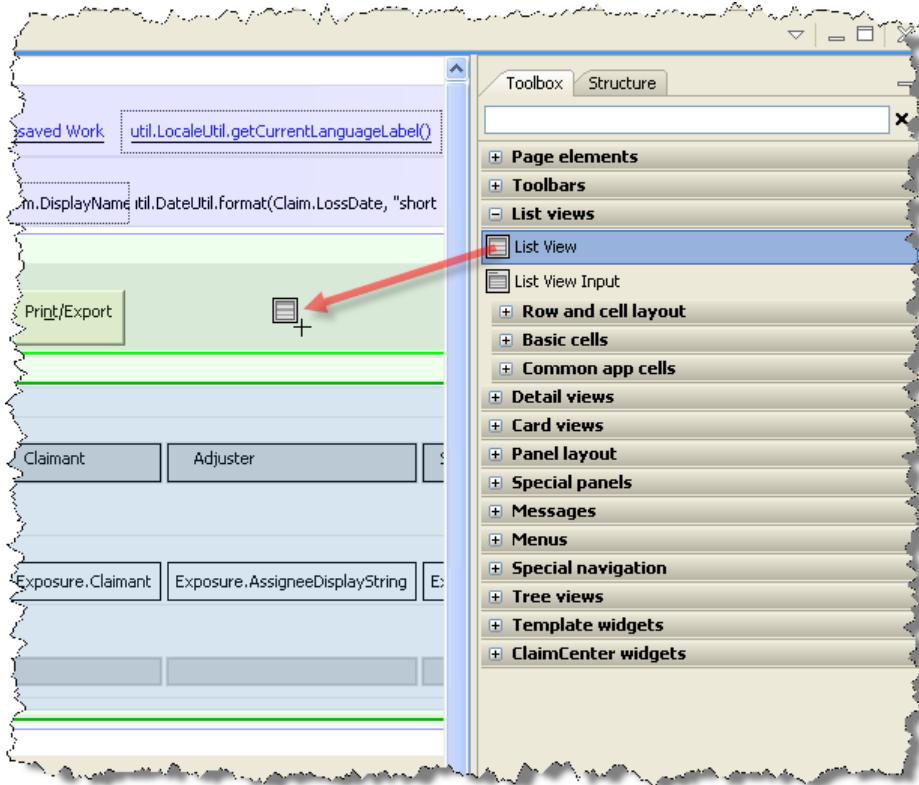
In most cases, data is viewed in list views and then edited in detail views. However, there are some places—for example, in the ClaimCenter financial transaction entry screens—in which it makes more sense to edit a list of items in place. For this purpose, you can make a list view editable so that you can add or remove rows, or modify cells of data.

The following is an example of a list view:

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

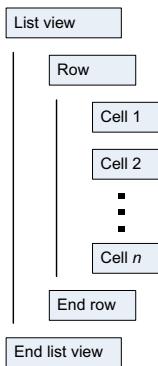
## Define a List View

Define a list view by dragging the **List View** element onto the PCF canvas. You can place the element anywhere a green line appears. For example:

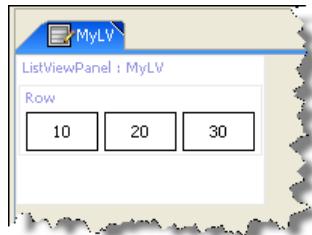


The `id` attribute is required; it identifies the panel so that it can be referenced by other PCF elements. The ID must be unique, and it must end with the text string `LV`.

A list view contains one or more *rows*, each containing one or more *cells*. The structure of the simplest one-row list view is illustrated below:



To define the rows and cells of the list view, use `Row` and `Cell` elements. Each occurrence of `Row` starts a new row, and each `Cell` creates a new column within the row. The following example creates a one-row, three-column list view:



The `id` attribute of a `Cell` element is required. It must be unique within the list view, but does not need to be unique across all of BillingCenter. 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 301.

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:

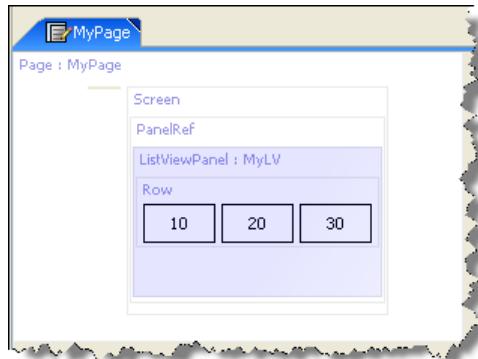
- Standalone
- Inline

### 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:

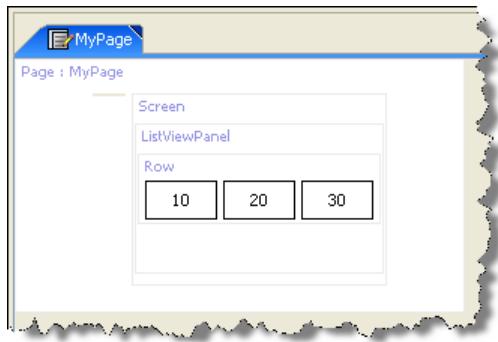


Set the `def` attribute of the `PanelRef` to the name of the list view; in this example, that is `MyLV`.

### 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.

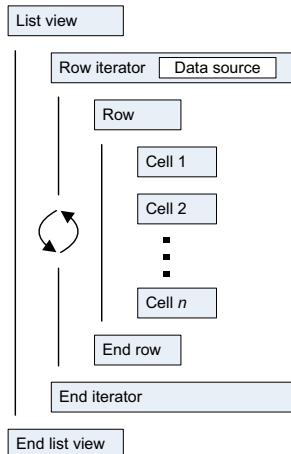
The following example defines an inline list view in a screen:



## Iterate a List View Over a Data Set

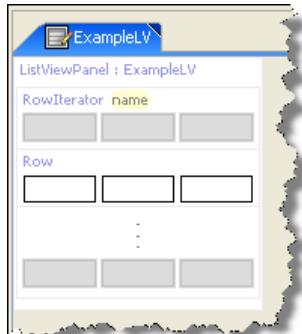
Most list views iterate over a data set and dynamically create a new row in the list for each record in the data set. The most common usage is showing an array of objects that belong to another object. For example, listing all activities that belong to a claim, or all users that belong to a group.

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.

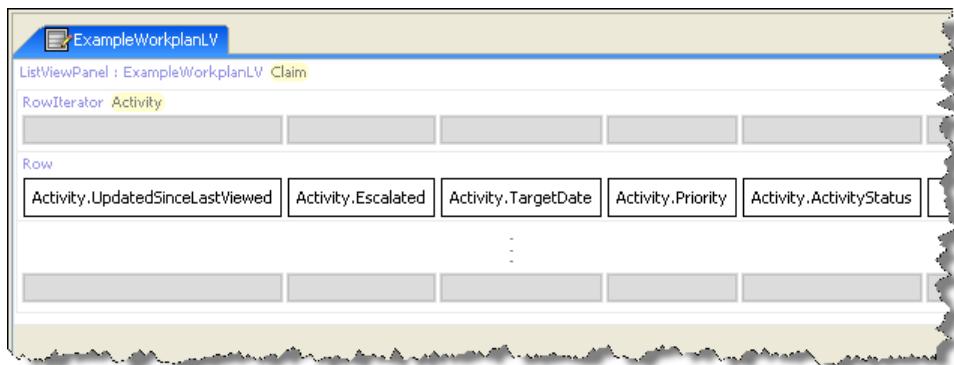
Define a row iterator with the **Row Iterator** PCF element. For example:



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 302.

The `elementName` attribute is the variable name that represents the “current” row in the list. You can use this variable anywhere within the row iterator as the root object that refers to the current row.

Consider the following example, in which a `Claim` variable represents a claim:



To iterate over the array of activities in the claim, this list view creates a row iterator whose `value` attribute is `Claim.Activities`. For each activity in this array, the iterator creates a row with multiple cells. The `elementName` attribute of the iterator is `Activity`; it represents the current row, and is used to get the values of the `Activity` object’s fields in each cell.

This example produces the following list view:

	Due	Priority	Status	Subject	Exposures	Exposure Type
	04/18/2008	Urgent	Open	Special Investigation Claim Escalation		
	01/29/2008	High	Open	Determine fault rating	(1) 1st Party Vehicle - Ray Newton	
	02/10/2008	High	Open	Mediation date		
	02/11/2008	High	Open	Trial date		

## Choose the Data Source for a List View

List views use different kinds of data sources to support different application requirements. The simplest data source is an array field on an entity type. An array field generally has a limited set of items that do not require a database query to retrieve. For example, the list of exposures for a claim is relatively short and is retrieved from the database as part of the overall claim, without a separate database query.

Other data sources for a list view involve a query and are more complex. This is especially true for search results or lists of items (activities, claims, and so on) on the Desktop. For example, a query as the source for a list view could be “all activities assigned to the current user that are due today or earlier.”

You specify the data source for a list view with the `value` property of the row iterator for the list view.

Source	Description
Array field	An <i>array field</i> on an entity type is identified in the <i>Data Dictionary</i> as an array key. For example, the <code>Officials</code> field on a <code>Claim</code> is an array key. Thus, you can define a list view based on <code>Claim.Officials</code> . In this case, each official listed on a specified claim is shown on a new row in the list view. You can also define your own custom Gosu methods that return array data for use in a list view. The method must return either a Gosu array or a Java list ( <code>java.util.List</code> ).
Query processor field	A <i>query processor field</i> on an entity type is identified in the <i>Data Dictionary</i> as a derived property returning <code>gw.api.database.IQueryBeanResult</code> . It represents an internally-defined query, and usually provides a more convenient and efficient way to retrieve data. For example, the <code>Claim.ViewableNotes</code> field performs a database query to retrieve only the notes on a claim that the current user has permission to view. This is more efficient than using the <code>Claim.Notes</code> array field, which loads both viewable and non-viewable notes and filtering the non-viewable ones out later.
Finder method	A <i>finder method</i> on an entity type is similar to a query processor field, except that it is not defined as field in the <i>Data Dictionary</i> . Instead, a finder method is an internally-defined Java class that performs an efficient query on instances of an entity type. For example, the <code>Activities</code> page of the <code>Desktop</code> uses a list view based on the finder method <code>Activity.finder.getActivityDesktopViewsAssignedToCurrentUser</code> .
Query builder result	A <i>query builder result</i> uses the result of an SQL query. For more information, see “Query Builder APIs” on page 129 in the <i>Gosu Reference Guide</i> .
Find expression query	A <i>find expression query</i> uses the result of an SQL query. Guidewire strongly recommends that you use query builder results as sources for list views instead of find expression queries.

List views behave differently depending on whether the source is an array or one of the query-backed sources.

Behavior	Array-backed list view	Query-backed list view
Loading data	The full set of data is loaded upon initially rendering the list view.	Only the data on the first page shown is fetched and loaded.
Paging	The full set of data is reloaded each time you move to a different page within the list view.	The query is re-run. Data is loaded only for the page that is viewable.
Sorting	The full set of data is reloaded each time the list view is sorted.	The query is re-run and sorted in the database. Therefore, you can sort only on columns that exist in the physical database, and not (for example) on virtual columns. Data is loaded only for the page that is viewable.
Filtering	The full set of data is reloaded each time the list view is filtered.	The query is re-run and filtered in the database. Therefore, you can filter only on columns that exist in the physical database, and not (for example) on virtual columns. Data is loaded only for the page that is viewable.
Editing	Paging, sorting, and filtering work as noted above, as long as any modified (but uncommitted) data is valid. Sorting and filtering can result in modified rows being sorted to a different page or filtered out of the visible list.	Paging, sorting, and filtering are disabled.
Best suited for	Short lists	Long lists
Additional notes	Do not use a query-backed editable list view in a wizard.	



# Location Groups

This topic provides an introduction to location groups.

This topic includes:

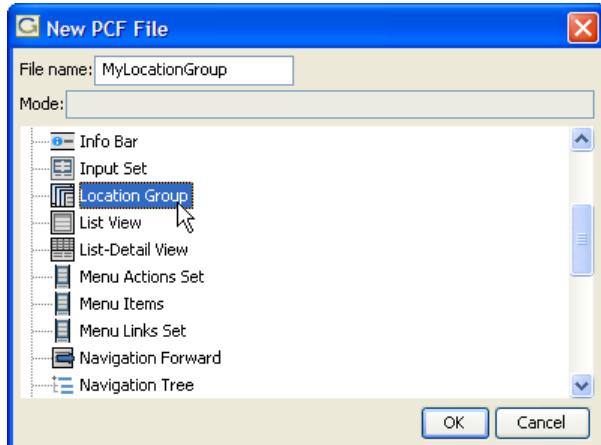
- “Location Group Overview” on page 305
- “Define a Location Group” on page 306
- “Location Groups as Navigation” on page 307

## Location Group Overview

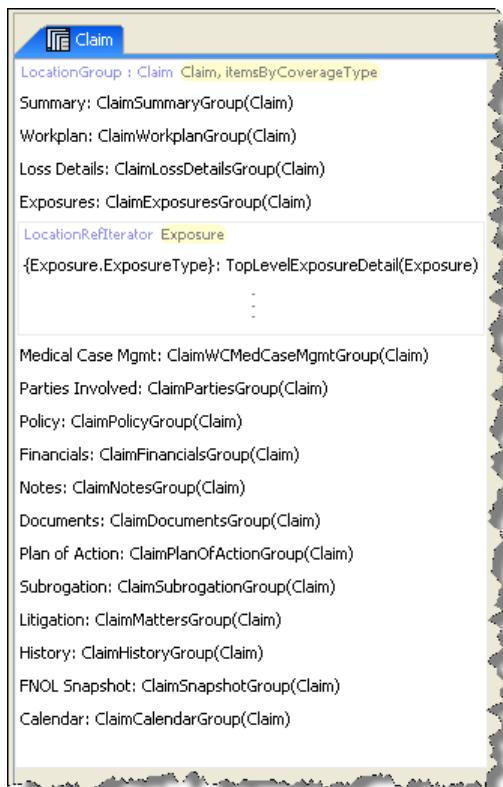
A *location group* is collection of locations. It is typically used to provide the structure and navigation for a group of related pages. BillingCenter can automatically display the appropriate menus and other interface elements that allow users to navigate among these pages.

## Define a Location Group

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



A location group must contain one or more references to another location. Any time that you navigate to the location group, BillingCenter 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

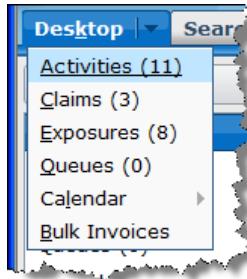
Depending on how a location group is used, BillingCenter displays menus and other screen elements for navigating to the locations within that group. Any time that you navigate to a location group, BillingCenter displays the first location within that group.

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

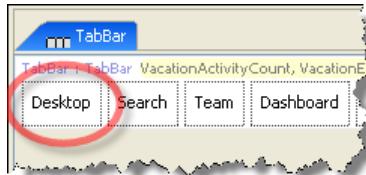
- Location Groups as Tab Menus
- Location Groups as Menu Links
- Location Groups as Screen Tabs

### 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:

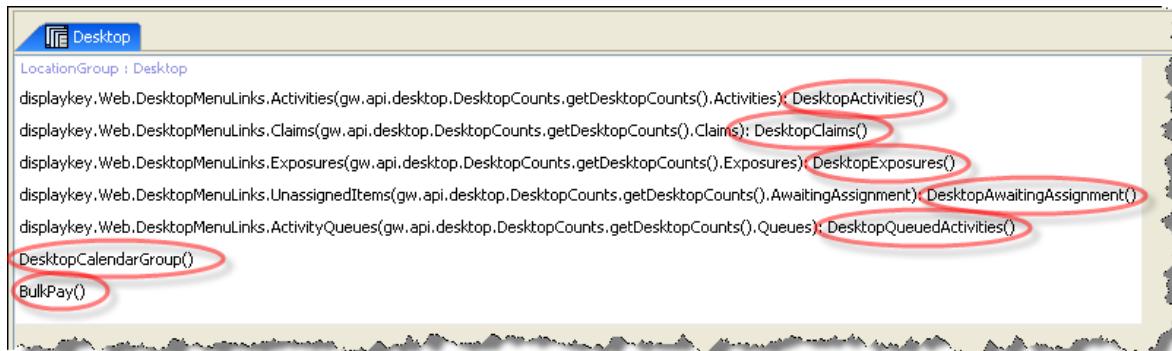


This tab is defined in the element named **TabBar** (under the **util** folder):



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:



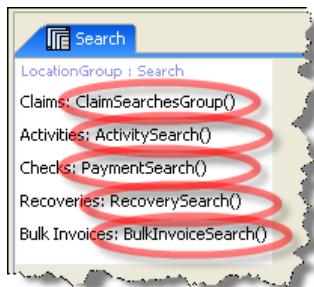
Inside this location group, there are multiple **Location Ref** elements defined, each one specifying a location. In this example, the locations referenced in the group correspond to the items in the **Desktop** menu. If the action for a tab is a location group containing more than one location, ClaimCenter 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 BillingCenter interface. As an example, consider the **Search** page in ClaimCenter, shown in the following diagram:



The following is the definition of the Search location group:

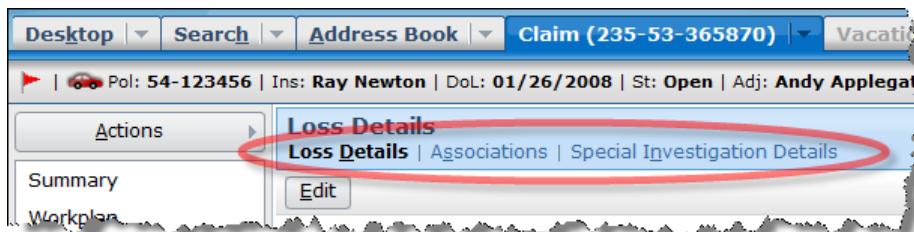


Inside this location group, there are multiple **Location Ref** elements defined, each one specifying a location.

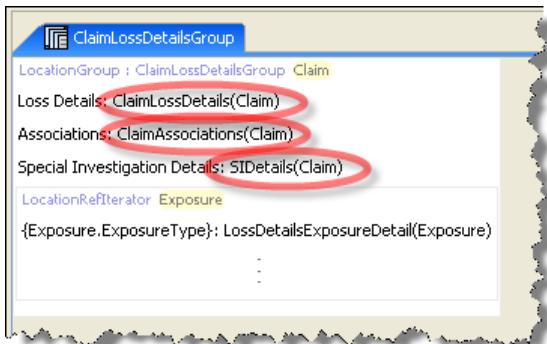
As ClaimCenter displays this location, it notices that it is a location group, and automatically creates menu links for each location within the group. Notice in this example that the **Location Ref** elements referenced in this group correspond to the items in the menu links.

## Location Groups as Screen Tabs

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



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



Inside this location group, there are multiple `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 BillingCenter user interface.

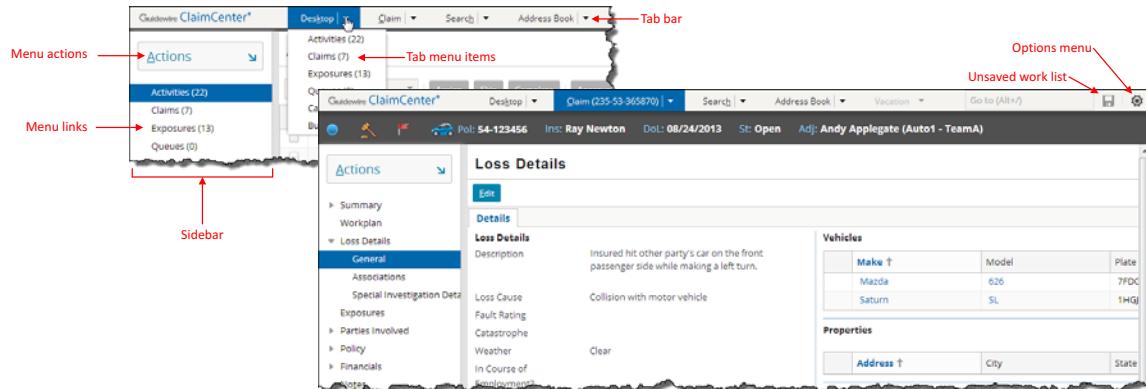
This topic includes:

- “Navigation Overview” on page 311
- “Tab Bars” on page 312
- “Tabs” on page 313

## 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:

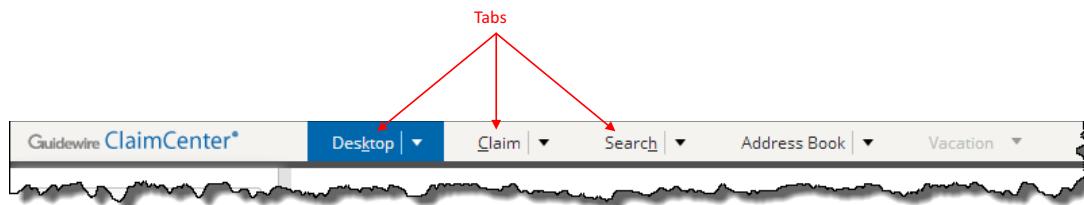


You can define the following types of navigation elements:

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 <b>Actions</b> menu in the sidebar that perform actions that are typically related to what you can do on the current tab.

## Tab Bars

A tab bar contains a set of tabs that run across the top of the application window, as in the following example:



You can do the following:

- Configure the Default Tab Bar
- Specify Which Tab Bar to Display
- Define a Tab Bar

You can also configure the individual tabs on a tab bar. For more information, see “Tabs” on page 313.

### Configure the Default Tab Bar

BillingCenter 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 BillingCenter 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, BillingCenter scans up the navigation hierarchy and checks whether a tab bar is explicitly set on a location group. If so, then that tab bar is used. If no tab bars are set, then the default tab bar is used.

For user interface clarity and consistency, we recommend that you set the tab bar only on the top-most location group in the hierarchy. However, a tab bar set on a child location group overrides the setting of its parent.

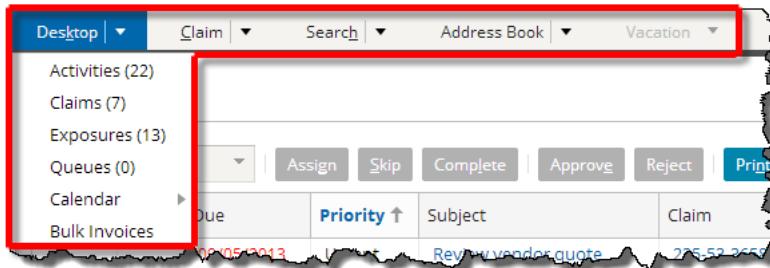
## Define a Tab Bar

Define a tab bar with the `TabBar` PCF element. For example:



## Tabs

Tabs are items in a tab bar that you can click on. A tab can be a single link that takes you directly to another location, it can be a drop-down menu, or it can be both. The following shows an example of tabs on a tab bar in ClaimCenter:



You can do the following:

- Define a Tab
- Define a Drop-down Menu on a Tab

### Define a Tab

Define a tab by placing a `Tab` PCF element with a `Tab Bar`. For example:



The `action` attribute of a tab defines where clicking the tab takes you. For example, to go to the Desktop location, set the `action` attribute to `Desktop.go()`.

### Define a Drop-down Menu on a Tab

A tab can contain a drop-down menu. As a tab has a menu, it shows the menu icon . Clicking this icon shows the menu items, while clicking the other parts of the tab performs the tab action.

Menu items on a tab are defined in the following ways:

- implicitly, using a location group
- explicitly, defined by `<MenuItem>` elements

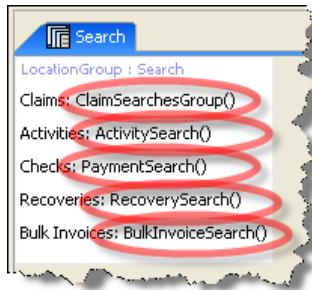
## Define a Tab Menu From a Location Group

As the `action` attribute of a tab is a location group, BillingCenter automatically creates menu items on the tab that correspond to the locations in that location group. For each location in the location group:

- a menu item is created in the tab
- the `label` attribute of the `Location Ref` is used as the label of the menu item
- the permissions of the location determine whether the menu item is available to the current user

For example, the action of the ClaimCenter **Search** tab goes to the **Search** location group. Its action attribute is defined as: `Search.Go()`.

This **Search** location group contains the `Location Ref` elements that appear as menu items on the tab:



This creates the menu items that appear on the **Search** tab:



## Define a Tab Menu Explicitly

You can create a menu on a tab by explicitly defining `Menu Item` elements within the `Tab` definition. This method of creating a menu supersedes the automatic menu items derived from the location group. If you build a menu explicitly, BillingCenter does not automatically add any other items to it.

# Configuring Search Functionality

BillingCenter provides a **Search** tab that you can use to search for specific entities. You can configure the **Search** tab to add new search criteria or modify or remove existing criteria. Configuring search functionality involves modifying the BillingCenter data model through metadata definition files and modifying the BillingCenter interface through page configuration files.

---

**WARNING** Guidewire strongly recommends that you consider all the implications before configuring the **Search** tab. Adding new search criteria can result in significant performance impacts, particularly in large databases. Guidewire recommends that you thoroughly test any search customizations for performance issues before you move them into a production database.

---

This topic includes:

- “BillingCenter Search Functionality” on page 315
- “Configuring BillingCenter Search” on page 316
- “Working with Search Criteria in XML” on page 317
- “Working with Search Criteria in Gosu” on page 320
- “The `SearchMethods` Class” on page 323
- “Search Criteria Validation Upon Server Start-up” on page 323

## BillingCenter Search Functionality

To search for a specific entity, select the **Search** tab from the BillingCenter interface. In the base configuration, you can search for the following:

- |                         |                              |
|-------------------------|------------------------------|
| • Accounts              | • Outgoing Producer Payments |
| • Activities            | • Payment Requests           |
| • Contacts              | • Payments                   |
| • Delinquency Processes | • Policies                   |

- Direct Bill Suspense Items
- Disbursements
- Invoices
- Producers
- Transactions
- Trouble Tickets

During a search, BillingCenter uses only those fields on the form for which you enter data. For example, if you search for a **Producer** and enter a **Producer Name** but not a **Producer Code**, BillingCenter omits **Producer Code** from the search.

For each search, BillingCenter uses one of the following types of objects to encapsulate search criteria.

Object type	Description
Virtual entity	BillingCenter implements most search criteria objects as <i>virtual entities</i> . A virtual entity has no underlying table in the BillingCenter database. Rather, these are non-persistent entities that exist only within the session in which you use them. An example of a non-persistent entity as a search criteria object is <code>DocumentSearchCriteria</code> , defined in <code>search-config.xml</code> .
Gosu class	An example of a Gosu class as a search criteria object is the <code>AccountSearchCriteria</code> class, which Guidewire defines in <code>AccountSearchCriteria.gs</code> .

Every field on the **Search** page maps to an attribute on the relevant search criteria entity. For example, in the **Account** search screen:

- Type maps to `AccountTypeCriterion` in the `AccountSearchDV` PCF file.
- `AccountTypeCriterion` maps to `searchCriteria.AccountType` in the same file.
- `AccountType` maps to `gw.search.AccountSearchCriteria`, the Gosu class definition that contains the business logic to search on the account type field.

## Configuring BillingCenter Search

There are multiple locations in BillingCenter code in which to configure search functionality. The particular file or files in which you configure search depends on the entity type that users can search. To configure search you must either modify `search-config.xml` or modify a Gosu search criteria class. The following table lists the entity types that users can search for, the corresponding search configuration type, and the specific file or files to modify.

Entity type to search	Search configuration type	Search configured in...
Account	Gosu class	<code>AccountSearchCriteria.gs</code>
AgencyBillMoneyRcvd	Gosu class	<code>PaymentSearchCriteria.gs</code> and <code>AgencyBillMoneyRcvdSearchStrategy.gs</code>
DelinquencyProcess	Gosu class	<code>DelinquencySearchCriteria.gs</code>
DirectBillMoneyRcvd	Gosu class	<code>PaymentSearchCriteria.gs</code> and <code>DirectBillMoneyRcvdSearchStrategy.gs</code>
Invoice	Gosu class	<code>InvoiceSearchCriteria.gs</code>
InvoiceItem	Gosu class	<code>InvoiceItemSearchCriteria.gs</code>
Payment	Gosu class	<code>PaymentSearchCriteria.gs</code>
Policy	Gosu class	<code>PolicySearchCriteria.gs</code>
PolicyPeriod	Gosu class	<code>PolicyPeriodSearchCriteria.gs</code>
Producer	Gosu class	<code>ProducerSearchCriteria.gs</code>
StatementInvoice	Gosu class	<code>StatementInvoiceSearchCriteria.gs</code>
SuspensePayment	Gosu class	<code>PaymentSearchCriteria.gs</code> and <code>SuspensePaymentSearchStrategy.gs</code>

Entity type to search	Search configuration type	Search configured in...
TransferTransaction	Gosu class	TransferTransactionSearchCriteria.gs
TroubleTicket	Gosu class	TroubleTicketSearchCriteria.gs
Contact	Gosu class	ABContactSearchCriteriaInfoEnhancement.gsx ContactSearchCriteriaEnhancement.gsx ContactManagerSystemPlugin.gs
AccountDisbursement Activity Address AgencyDisbursement Charge CollateralDisbursement CommissionPlan Credit DirectSuspPmntItem Document NegativeWriteoff Note OutgoingProducerPmnt PaymentRequest SuspenseDisbursement SuspensePayment Transaction Writeoff	Virtual entity	search-config.xml

For information on configuring search for contacts, see “Searching for Contacts” on page 83 in the *Contact Management Guide*.

## Limiting Search Results

The maximum number of search results can be limited on most BillingCenter search screens.

### Search Configuration Parameters in BillingCenter

The following parameters help make search screens perform better by limiting the number of search results that BillingCenter will display:

- **MaxSearchResults** – Maximum number of search results that BillingCenter returns in a search. Defined as a configuration parameter in config.xml. See “MaxSearchResults” on page 57.
- **MaxContactSearchResults** – Maximum number of contacts that BillingCenter returns in a search. Defined as a configuration parameter in config.xml. See “MaxContactSearchResults” on page 57.

### Search Configuration PCF Property in BillingCenter

You can edit the following PCF property for search screens that use the MaxSearchResults configuration parameter, in order to override MaxSearchResults for one search screen:

- **maxSearchResults** – Maximum number of search results that BillingCenter returns in a search from this search screen only. Defined as an xml attribute in the PCF.

### See also

For details, see “Search Parameters” on page 56.

## Working with Search Criteria in XML

You use the search-config.xml file to define a mapping between the key data entities and certain non-persistent entities used for search criteria. The entries in the file have the following basic structure.

```
<CriteriaDef entity="name" targetEntity="name">
  <Criterion property="attributename" targetProperty="attributename" matchType="type"/>
```

&lt;/CriteriaDef&gt;

The following table describes the XML elements in the `search-config.xml` file.

Element name	Subelement	Description
SearchConfig	CriteriaDef	Root element in <code>search-config.xml</code> .
CriteriaDef	Criterion	Specifies the mapping from a search criteria entity to the target entity on which to search.  <b>WARNING</b> Do not add new <code>CriteriaDef</code> elements to <code>search-config.xml</code> . Instead, modify only the contents of existing <code>CriteriaDef</code> elements.
Criterion		Specifies how BillingCenter matches a column (field) on the search criteria to the query against the target entity. Use this element to perform simple matching only. Simple matches are criteria that match values in a single column of the same type in the target entity.

#### See also

- “The `<CriteriaDef>` Element” on page 318.
- “The `<Criterion>` Subelement” on page 319.

## The `<CriteriaDef>` Element

A `<CriteriaDef>` element specifies the mapping from a search criteria entity to the target entity on which to search. For example, a `<CriteriaDef>` element can specify a mapping between a `DocumentSearchCriteria` entity and a `Document` entity. A `<CriteriaDef>` element uses the following syntax.

```
<CriteriaDef entity="entityName" targetEntity="targetEntityName">
```

These attributes have the following definitions.

<code>&lt;CriteriaDef&gt;</code> attribute	Required	Description
entity	Yes	Type name of the criteria entity
targetEntity	Yes	Type name of the target entity.

It is also possible to map a single search criteria entity to more than one target entity. For example, the `ClaimSearchCriteria` object has a `<CriteriaDef>` element associated with all of the following entities:

- `PolicyPeriod`
- `Producer`
- `ProducerCode`

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 BillingCenter installation.

**WARNING** Guidewire strongly recommends you do not remove `<CriteriaDef>` elements that exist in the base configuration.

A `<CriteriaDef>` element can have the following subelements.

<code>&lt;CriteriaDef&gt;</code> subelement	Description
Criterion	Performs simple, one-to-one mapping between a criteria entity attribute and a target entity attribute.

## The <Criterion> Subelement

Within a <CriteriaDef> element you can define zero or more <Criterion> subelements. A <Criterion> element performs simple, one-to-one mapping between a criteria entity attribute and a target entity attribute. A <Criterion> element uses the following syntax.

```
<Criterion property="attributename"
           targetProperty="attributename"
           forceEqMatchType="booleanproperty"
           matchType="type"/>
```

These attributes have the following definitions.

<Criterion> attribute	Required	Description
property	•	The name attribute on the criteria entity. BillingCenter uses this value to get the user's search term from the criteria entity.
matchType	•	This attribute is dependent on the data type of the targetProperty. See the following table for possible values.
forceEqMatchType		<p>The name of a Boolean property on the criteria entity:</p> <ul style="list-style-type: none"> <li>If this attribute evaluates to true, then the Criterion uses an eq (equality) match.</li> <li>If this attribute evaluates to false, then the Criterion uses the matchType that the Criterion specifies to perform the match.</li> </ul> <p>For example:</p> <pre>&lt;Criterion property="StringProperty"            forceEqMatchType="FlagProperty"            matchType="startsWith"/&gt;</pre> <p>This code uses a startsWith match for StringProperty unless the FlagProperty on the criteria entity is true, in which case, the match uses an eq match type.</p>
targetProperty		<p>The name attribute on the entity on which to search.</p> <p><b>IMPORTANT</b> Do not use a virtual property on the entity as the search field.</p>

The following list describes the valid matchType values. For String objects, matchType case-sensitivity depends on the database, except for startsWith and contains, which are always case-insensitive.

Match type	Evaluates to	Use with data type	Comments
contains		String	<p><b>IMPORTANT</b> Guidewire strongly recommends that you avoid using the contains match type, if at all possible. The contains match type is the most expensive type in terms of performance.</p>
eq	equals	Numeric or Date	
ge	greater than or equal	Numeric or Date	
gt	greater than	Numeric or Date	
le	less than or equal	Numeric or Date	
lt	less than	Numeric or Date	
startsWith		String	<p>The startsWith match type is very expensive in terms of performance, second only to the contains match type. Use startsWith with caution.</p>

## Performance Tuning for Specific Search Criteria

It is possible that adding an index can improve performance. The exact index to add depends on the database that you use and the details of the situation. Whenever you change the search criteria by adding or modifying a <Criterion> subelement, be certain that appropriate indexes are in place. Guidewire recommends that you consult a database expert.

For example, suppose that you add a column that is the most restrictive equality condition in your search implementation. In this case, consider adding an index with this column as the leading key column.

**IMPORTANT** For performance reasons, Guidewire strongly recommends that you avoid the contains match type if at all possible. The contains match type is the most expensive type in terms of performance.

## Do Not Attempt to Modify the Required Search Properties

Guidewire divides the main search screens into required and optional sections. Guidewire has carefully chosen the properties in the required section to enhance performance. Therefore, do not change which properties are required properties. Adding your own required search criteria can cause performance issues severe enough to bring down a production database.

In addition, Guidewire has carefully chosen the match types of the existing required properties, due to restrictions on configuring fields on tables that are joined to the search table. Therefore, do not change the match types of existing required fields.

**WARNING** For performance reasons, Guidewire expressly prohibits the addition of new required fields or changing the match type of existing required fields in the BillingCenter search screens.

## Working with Search Criteria in Gosu

In the base BillingCenter configuration, Guidewire provides Gosu classes to configure database search for a number of entity types. The following table lists some entity types for which users can search and the Gosu classes that you modify to configure the user interface for that type of search.

Entity type to search	Gosu search criteria class
Account	gw.search.AccountSearchCriteria
AgencyBillMoneyRcvd	gw.search.PaymentSearchCriteria
DelinquencyProcess	gw.search.DelinquencySearchCriteria
DirectBillMoneyRcvd	gw.search.PaymentSearchCriteria
Invoice	gw.search.InvoiceSearchCriteria
Payment	gw.search.PaymentSearchCriteria
Policy	gw.search.PolicySearchCriteria
Producer	gw.search.ProducerSearchCriteria
SuspensePayment	gw.search.PaymentSearchCriteria
TroubleTicket	gw.search.TroubleTicketSearchCriteria

Guidewire also provides a Gosu enhancement that you can use to configure searches for Contact objects.

`gw.plugin.contact.impl.ContactSearchCriteriaEnhancement.gs`

For more information on BillingCenter support for Contact searches, see “BillingCenter Support for Contact Searches” on page 107 in the *Contact Management Guide*.

For a complete list of entity types for which you can configure search criteria in Gosu, see “Configuring BillingCenter Search” on page 316.

Because these Gosu classes and enhancements are public, you are free to modify them as you want.

**Note:** To improve contact search, Guidewire adds the following fields to the **Contact** entity as denormalization fields from the **Address** entity:

- **CityDenorm**
- **Country**
- **PostalCodeDenorm**
- **State**

**WARNING** Guidewire strongly recommends that you consider all the implications before customizing any search configuration object. Adding new search criteria can result in significant performance impacts, particularly in large databases. Guidewire recommends that you thoroughly test any search customizations for performance issues before you move them into a production database.

## Example: Modifying the PolicySearchCriteria Class

You can add additional search criteria to policy searches by modifying the **PolicySearchCriteria** class.

1. Extend the **PolicyPeriod** entity and add your search field to it as an extension column. The example adds a vehicle VIN column to the **PolicyPeriod** extension **PolicyPeriod.etx**.
2. Modify the **PolicySearchCriteria** class by adding the new search criteria. Declare a variable that will map to a field in the PCF.
3. Add an additional search criteria field in **PolicySearchScreen.pcf** that maps to the variable in the **PolicySearchCriteria** class. The example modifies this PCF file and adds an optional Input widget for the VIN number. BillingCenter displays this VIN field only if you perform a Policy search on a **SearchObjectType** of **Policy**.
4. Incorporate the variable into the query method defined in the **PolicySearchCriteria** class.

The following table lists the BillingCenter files involved in modifying a base configuration Policy search.

Location	Contains...
<b>PolicySearchDV.pcf</b>	Contains a set of fields that accept search criteria related to policies, such as policy number, account number, and billing code. There is also a set of fields for searching for contacts, with search criteria like company name, first name, last name, address, and so on. These fields all take <b>searchCriteria</b> values, like <b>searchCriteria.AccountNumber</b> and <b>searchCriteria.ContactCriteria.LastName</b> .
<b>PolicyPeriod.etx</b>	Extends the entity <b>PolicyPeriod.eti</b> . In the example, you add a VIN column.
<b>gw.search.PolicySearchCriteria.gs</b>	Defines the Gosu class used for the search. It has a set of private methods that begin with <b>restrictSearchBy</b> . These methods specify various search fields and what kind of search to use on them. If you modify the search fields, you need to either modify the appropriate <b>restrictSearchBy</b> method or add a new <b>restrictSearchBy</b> method.

### To modify the Policy search configuration

The following example adds a vehicle VIN field to the BillingCenter **Search Policies** screen.

1. Open BillingCenter Studio and navigate **configuration** → **config** → **Metadata** → **Entity**

**2.** Right-click the file `PolicyPeriod.eti`, and choose **New → Entity Extension**.

**3.** In the extension file, `PolicyPeriod.etcx`, add the following column.

```
<column
    desc="VIN (vehicle identification number) of the vehicle."
    name="Vin"
    type="vin"/>
```

**4.** Save the file.

**5.** Press **Ctrl+N** and enter `PolicySearchCriteria` to find `gw.search.PolicySearchCriteria`.

**6.** Open `gw.search.PolicySearchCriteria` for editing and add the following variable to the list of variables after the class declaration.

```
var _vin: String as VIN
```

If you see a message asking if you want to edit the class, click **Yes**.

**7.** Add the following code to the method `restrictSearchByPolicyFields`.

```
if (VIN.NotBlank) {
    query.compare("VIN", Equals, VIN)
}
```

**8.** Save your changes.

**9.** Add a display key for the VIN field label:

**a.** Navigate to **configuration → config → Localizations → lang** in Studio, and open the `display.properties` file.

**b.** Locate the display key entries that begin with `PolicySearchDV`, and add the following line.

```
Web.PolicySearchDV.Vin = VIN
```

**10.** Press **Ctrl+N** and enter `PolicySearchDV`, and then click `PolicySearchDV.pcf` to open this PCF file.

**11.** Drag an **Input** widget from the **Toolbox** on the right and drop it under the **Billing Method** field.

If you see a message asking if you want to edit the PCF file, click **Yes**.

**12.** Enter the following values for this widget in the **Properties** area at the bottom of the screen.

<code>editable</code>	<code>true</code>
<code>id</code>	<code>VINCriterion</code>
<code>label</code>	<code>displaykey.Web.PolicySearchDV.Vin</code>
<code>required</code>	<code>false</code>
<code>value</code>	<code>searchCriteria.VIN</code>

**13.** Save your changes.

**14.** Stop and restart BillingCenter.

**15.** Test your work by navigating to the **Search Policies** screen in BillingCenter and entering various search criteria.

## Enabling Searches on the Address Field

Performing a search based on the first Address field is disabled by default. To enable searches on the Address field, modify the `AddressOwnerFieldId` Gosu class as shown below. Afterward, searches based on the first Address field will be enabled in the Account, Policy, Producer, Invoice, Trouble Ticket, Delinquency, and Agency Money Receipts screens.

**1.** Open BillingCenter Studio.

**2.** Press **Ctrl+N** and enter `AddressOwnerFieldId` to load the Gosu source file.

**3.** Locate the `HIDDEN_FOR_SEARCH` variable.

4. Remove the ADDRESSLINE1 value from the variable's list of possible values.
5. Save the file.
6. Stop and restart BillingCenter.

## The SearchMethods Class

BillingCenter provides a search-related class, `gw.search.SearchMethods`, that you can use to validate the input fields for the associated search screen. The class contains a number of methods, each of which validates the input fields on a search screen and determines whether a search field is `null` or not.

For example:

```
public static function validateAndSearch(searchCriteria : ChargeSearchCriteria,  
    isClearBundle : Boolean) : ChargeQuery {  
    if (!User.util.CurrentUser.UnrestrictedUser and searchCriteria.Account == null and  
        searchCriteria.PolicyPeriod == null) {  
        throw new DisplayableException(displaykey.Java.Search.Error.RequiredNotPresent)  
    }  
    else {  
        return searchCriteria.performSearch(isClearBundle)  
    }  
}
```

In this method, if both `Account` and `PolicyPeriod` are `null`, the method returns a `RequiredNotPresent` error, which is a standard display key.

## Search Criteria Validation Upon Server Start-up

The BillingCenter 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:

- That the `CriteriaDef` entity and `targetEntity` attributes reference real entities.
- That the `targetEntity` type is a persistent entity.
- That the `targetProperty` attributes reference searchable properties on the target entity, except for those on `ArrayCriterion` elements that must reference an array column.
- That the type of each `property` attribute on a `Criterion` element matches the type of its corresponding `targetProperty`. For example, that they are both strings or both numbers.
- That all `Criterion` match types (`matchType`) are suitable for the criterion property. For example, you can use `startsWith` only for `string` properties.
- All `CriterionChoice` property (`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.
- That all `Option` label attributes reference valid display keys.
- All `ArrayCriterion` `arrayMemberProperty` attributes reference searchable properties on the array member entity.

T



# Configuring Special Page Functions

This topic describes how to configure special functionality related to pages.

This topic includes:

- “Adding Print Capabilities” on page 325
- “Adding Print Capabilities” on page 325
- “Linking to a Specific Page: Using an EntryPoint PCF” on page 327
- “Linking to a Specific Page: Using an ExitPoint PCF” on page 329

**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 BillingCenter interface. This section explains the print capabilities and how to use them. It covers the following topics:

- Overview of the Print Functionality
- List View Printing

### Overview of the Print Functionality

You can use the BillingCenter printing functionality to print the list view parts of the data visible on a BillingCenter screen. You can control both the output format and which list view objects are printed. Most commonly, a page prints as PDF. However, BillingCenter 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 BillingCenter you can print object lists such as the **Activities** list on the **Desktop**, for example.

All client machines must have a supported version of the Acrobat Reader available to support PDF printing.

## Configuration Parameters Related to Printing

The following optional print parameters in `config.xml` control the default print settings globally. For information on configuration parameters, see “Application Configuration Parameters” on page 27.

<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 the following location:

<http://www.w3.org/TR/REC-CSS2>

## Security Related to Printing

BillingCenter provides a `1vprint` system permission that you can use to print the information that appears in a list view. In the base BillingCenter configuration, Guidewire marks this system permission as retired. As it is retired, BillingCenter does not enable this permission in any of the base configuration system roles.

To provide this functionality, do the following:

1. First, un-retire the permission by editing the `SystemPermissionType` typelist in Guidewire Studio and changing the `Retired` field from `true` to `false`.
2. Second, specifically add this permission to a BillingCenter role.
3. Finally, assign this role to an individual user.

## Gosu API Methods for Printing

BillingCenter has a Gosu `gw.api.print` API that contains a number of print-related methods. Guidewire recommends that you avoid using this API in your print configurations with the exception of the following methods:

- `ListViewPrintOptionsPopupAction`
- `PrintSettings`

## List View Printing

You use list view printing to output a list in either a PDF format or a CSV (comma-separated value) format. You can interactively choose the type of output delivered by the Print button. This type of printing uses the ToolbarButton element with an action attribute. The action attribute contains a Gosu expression that calls the Gosu API ListViewPrintOptionsPopupAction method. The following example illustrates a list view printing method:

```
<ToolbarButton label="displaykey.Java.ListView.Print" id="PrintButton"
    action="gw.api.print.ListViewPrintOptionPopupAction.printListViewWithOptions('MyListView')"/>
```

If you choose to print in CSV format, you can also choose which columns to print. For example, you can use list view printing to print the Desktop Activities page.

## Linking to a Specific Page: Using an EntryPoint PCF

It is possible to connect directly to BillingCenter using a URL that leads to a specific BillingCenter page. You can define your own links or entry points. Thus, if the BillingCenter server receives a connect request from an external source and the request has both the correct format and parameters, BillingCenter serves the requested page.

In the base configuration, BillingCenter provides a number of EntryPoint PCF examples. You can find these in the following location in Studio:

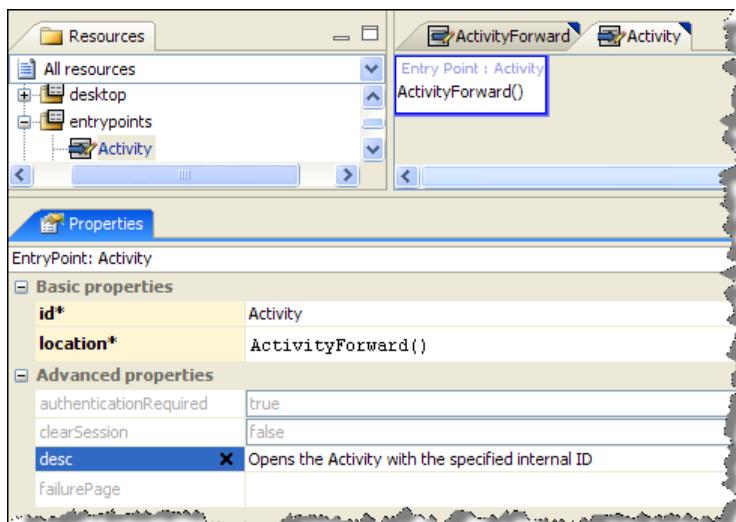
configuration → config → Page Configuration → pcf → entrypoints

These PCF pages are examples only. If you use one, you must customize it to meet your business needs. You can also use them as starting points for your own EntryPoint PCF pages.

## Entry Points

An entry point takes the form of a URL with a specific syntax. The entry URL specifies a location that a user enters into the browser. If the BillingCenter server receives a connection request with a specific entry point, BillingCenter responds by serving the page based on the entry point configuration.

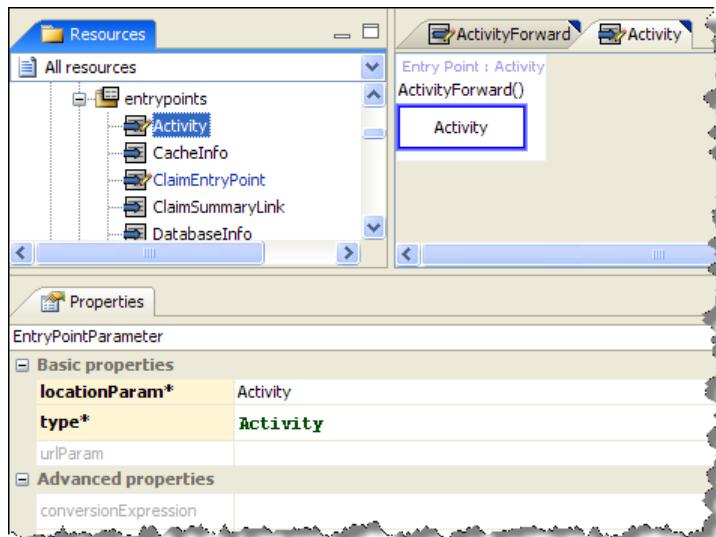
To implement this functionality, you must create an EntryPoint PCF (in the `entrypoints` folder). The following graphic illustrates an EntryPoint PCF.



The EntryPoint PCF contains the following parameters:

<code>authenticationRequired</code>	Specifies that BillingCenter must authenticate the user before the user can access the URL. If true, BillingCenter requires that the user already be authenticated to enter. If the user is not already logged in, BillingCenter presents a login page before rendering the entry point location. The default is true. Guidewire strongly recommends that you think carefully before setting this value to false.
<code>clearSession</code>	If true, clears the server session for this user as the user enters this entry point
<code>desc</code>	Currently, does nothing.
<code>failurePage</code>	Specifies the page to send the user if BillingCenter can not display the entry point. Failures typically happen any time that the data specified by the URL does not exist. The default is Error.
<code>id</code>	Required. The BillingCenter 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, BillingCenter, as the URI.  For example, if the URI is XXX, then it is possible to enter the application at <code>http://myserver/myapp/XXX.do</code> .
<code>location</code>	Required. The ID of the page, Forward, or wizard to which you want to go. Guidewire recommends that if you want the entry point to perform complex logic, use a Forward.  See "To create a forwarding EntryPoint PCF" on page 329 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 BillingCenter uses to convert a URL parameter to the value passed to the location parameter.
<code>desc</code>	Currently, does nothing.
<code>locationParam</code>	Required. The name of the LocationParameter on the EntryPoint target location that this parameter sets.
<code>optional</code>	Specifies whether the parameter is optional. If set to true, BillingCenter does not require this parameter.
<code>type</code>	Required. Specifies what type to cast the incoming parameter into, such as String or Integer.
<code>urlParam</code>	The name of the parameter passed with the URL. For example, if the urlParam is Activity and the entry point URI is ActivityDetail, you would pass Activity 3 as:  <code>http://myserver/myapp/ActivityDetail.do?Activity=3</code>

## Creating a Forwarding EntryPoint PCF

A *forward* is a top-level PCF location element similar to a page or wizard. However, it has no screen. It merely forwards you to another location. You define a forward separately from the **EntryPoint** PCF. However, you set the forward for a PCF in the **EntryPoint** PCF location attribute.

**Note:** For an example of how to define a forward, see `AccountForward` in BillingCenter Studio at `pcf → account → AccountForward`.

### To create a forwarding EntryPoint PCF

1. Define a separate entry point (PCF) with `authenticationRequired` property set to `false`. This PCF is effectively a forwarding page to handle the seamless login.
2. Set the `location` attribute of the entry point to use a `Forward` to call the `AuthenticationServicePlugin`.
3. Do one of the following:
  - If the plugin login is successful, forward the user onto the actual page (the desktop, for example) to which you intended to send the user in the first place. (This is the page to which the user would have gone if `authenticationRequired` had been set to `true`.)
  - If the plugin login is not successful, redirect the user to an error page or an alternate login page.

Suppose that there are several destinations to which you wish the user to go. In this case, consider passing a parameter to the entry point forward, so you can have the seamless login logic all in that one place.

## Linking to a Specific Page: Using an ExitPoint PCF

It is possible to create a link from a BillingCenter 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 BillingCenter screen.

In the base configuration, BillingCenter provides a number of **ExitPoint** PCF examples. You can find these in the following location in Studio:

`configuration → config → Page Configuration → pcf → exitpoints`

**Note:** These PCF pages are examples only. If you use one, Guidewire expects you to customize it to meet your business needs. You can also use them as starting points for your own **ExitPoint** PCF pages.

### Creating an ExitPoint PCF

The following example takes you through the process of creating a new exit point PCF and then modifying a BillingCenter 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 BillingCenter. You can also configure the **ExitPoint** PCF functionality to log out the user or to possibly reuse the current window.

### Step 1: Create the ExitPoint PCF File.

The first step is to create a new `ExitPoint` PCF file and name it `AnyURL`.

1. Within Studio, navigate to `configuration` → `config` → `Page Configuration` → `pcf` → `exitpoints`, and then select `New` → `PCF File` from the right-click menu.
2. Enter `AnyURL` for the file name in the `New PCF File` dialog and select `Exit Point` as the file type.
3. Select the `AnyURL` file, so that Studio outlines the `ExitPoint` element in blue.
4. Select the `Properties` tab at the bottom of the screen and set the listed properties. This example pushes the URL to a popup window that leaves the user logged into BillingCenter. 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 BillingCenter 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 BillingCenter 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 BillingCenter data model, which would require a server restart.
2. Log into BillingCenter 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. BillingCenter presents a success message after it reloads the PCF files from the local file system.
5. Log into BillingCenter 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 BillingCenter opens a popup window and loads the URL that you set on the **constructMyURL** function. If you followed the steps of this example exactly, BillingCenter loads the Guidewire Internet home page into the popup window.



# Workflow and Activity Configuration



# Using the Workflow Editor

This topic covers basic information about the workflow editor in Guidewire Studio.

This topic includes:

- “Workflow in Guidewire BillingCenter” on page 335
- “Workflow in Guidewire Studio” on page 336
- “Understanding Workflow Steps” on page 337
- “Using the Workflow Right-Click Menu” on page 338
- “Using Search with Workflow” on page 338

## Workflow in Guidewire BillingCenter

Guidewire BillingCenter uses workflow to drive some of its payment processes. Guidewire defines and stores each base configuration workflow process as a separate file in the following directory:

*BillingCenter/modules/bc/config/workflow*

Each file name corresponds to the workflow process that it defines (for example, *CancelImmediately.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:

*BillingCenter/modules/configuration/config/workflow*

### See also

- “BillingCenter Workflows and Delinquency Plans” on page 413.
- For information on workflow structure and design, see “Guidewire Workflow” on page 341.

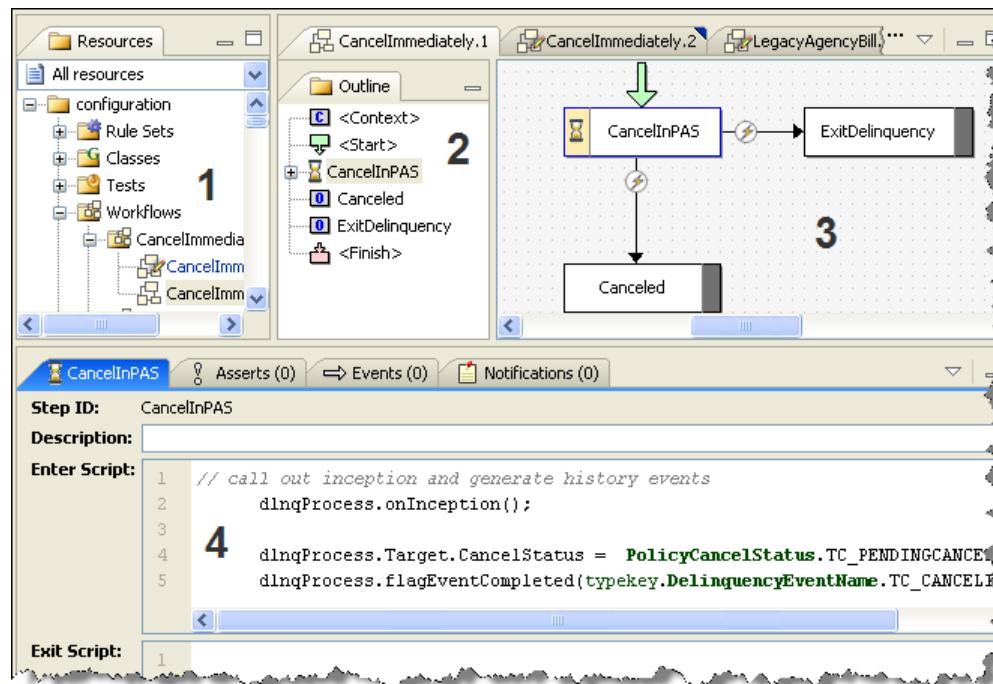
## Workflow in Guidewire Studio

Even though Guidewire defines the workflow scripts in XML files, you use Guidewire Studio to view, edit, manage, and create new workflows scripts. Thus, you do not work directly with XML files. Instead, you work with their representation in Guidewire Studio, in the Studio **Workflows** editor.

To access the workflow editor, navigate to **configuration** → **config** → **Workflows**, and then select a workflow. Within the **Workflows** editor, there are multiple work areas, each of which performs a specialized function:

Area	View	Description
1	Tree view	Studio displays each workflow type as a node in the <b>Resources</b> tree. If you have multiple versions of a workflow type, Studio displays each one with an incremental version number at the end of the file name.
2	Outline view	Studio displays an outline of the selected workflow process in the <b>Outline</b> 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 <b>Workflow</b> 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 BillingCenter base configuration, Guidewire defines a **CancelImmediately** 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 346
<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 347
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 349
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 350
ActivityStep		An ActivityStep is similar to an AutoStep, except that it can use any of the branch types, such as a TRIGGER or a TIMEOUT, to move to the next step. However, before an ActivityStep branches to the next step, it waits for one or more activities to complete.	"ActivityStep" on page 351
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 352
GO		Indicates a branch or transition to another workflow step. It occurs only within an AutoStep workflow step. <ul style="list-style-type: none"> <li>• If there is only a single GO element within the workflow step, branching occurs immediately upon workflow reaching that point.</li> <li>• 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.</li> </ul>	"GO" on page 355
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 356
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 358
Outcome		Indicates a possible outcome for the workflow. This step is special. It indicates that it is a last step, out of which no branch leaves.	"Outcome" on page 353
<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 347

## 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. BillingCenter 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 346.

Type	Workflow contains	Icon	Step	Description
AutoStep	Zero, one, or more		 Step1	Step that BillingCenter guarantees to finish immediately. See “AutoStep” on page 349.
ManualStep	Zero, one, or more		 Step2	Step that waits for an external TRIGGER to occur or a TIMEOUT to pass. See “ManualStep” on page 352.
ActivityStep	Zero, one, or more		 Step3	Step that waits for one or more activities to complete before continuing. See “ActivityStep” on page 351.
MessageStep	Zero, one, or more		 Step4	Special-purpose step designed to support messaging-based integrations. See “MessageStep” on page 350.
Outcome	One or more		 Outcome	Special final step that has no branches leading out of it. See “Outcome” on page 353.

## 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 <b>Rename</b> from the right-click menu. This opens the <b>Rename StepID</b> dialog in which you can enter the new step name.
To change a workflow step type	Select <b>Change Step Type</b> 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 <b>Move Up</b> ( <b>Move Down</b> ) 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 <b>New &lt;BranchType&gt;</b> 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 <b>Delete</b> from the right-click menu. This action removes the workflow step from the workflow outline. The workflow editor does not permit you to remove the workflow step that you designate as the workflow start step.

### See also

- To learn how to localize names of workflow steps, see “Localizing Guidewire Workflow” on page 71 in the *Globalization Guide*.

## Using Search with Workflow

It is possible to search for a specific text string within a workflow by selecting **Find in Path** from the Studio **Edit** menu. You can search on a localized text strings as well. You can also select a workflow and select **Find in Path** from the right-click menu.

- If you use the **Search** menu option, you can filter the resources to check.
- If you use the right-click menu option, then the search encompasses all active resources.

In either case, Studio opens a search pane at the bottom of the screen and displays any matches that it finds. You can click on a match to open the workflow in which the match exists.



# Guidewire Workflow

This topic covers BillingCenter workflow. Workflow is the Guidewire generic component for executing custom business processes asynchronously.

This topic includes:

- “Understanding Workflow” on page 341
- “Workflow Structural Elements” on page 346
- “Common Step Elements” on page 347
- “Basic Workflow Steps” on page 349
- “Step Branches” on page 354
- “Creating New Workflows” on page 359
- “Instantiating a Workflow” on page 363
- “The Workflow Engine” on page 365
- “Workflow Subflows” on page 368
- “Workflow Administration” on page 369
- “Workflow Debugging, Logging, and Testing” on page 370

## Understanding Workflow

There are multiple ways to think about workflow:

Term	Definition
workflow, workflow instance	A specific running instance of a particular business process. Guidewire persists a workflow instance to the database as an entity called <code>Workflow</code> .
workflow type	A single kind of flow process, for example, a Cancellation workflow.
workflow process	A definition of a workflow type in XML. Guidewire defines workflow processes in XML files that you manage in Guidewire Studio through the graphical <code>Workflows</code> editor.

Discussions about *workflow* in general or the *workflow system* refer usually to the workflow infrastructure as a whole.

## Workflow Instances

Think of a *workflow instance* as a row in the database marking the existence of a single running business flow. BillingCenter creates a workflow instance in response to a specific need to perform a task or function, usually asynchronously. For example, in the base configuration, BillingCenter uses workflows to manage agency billing, and account and policy delinquencies.

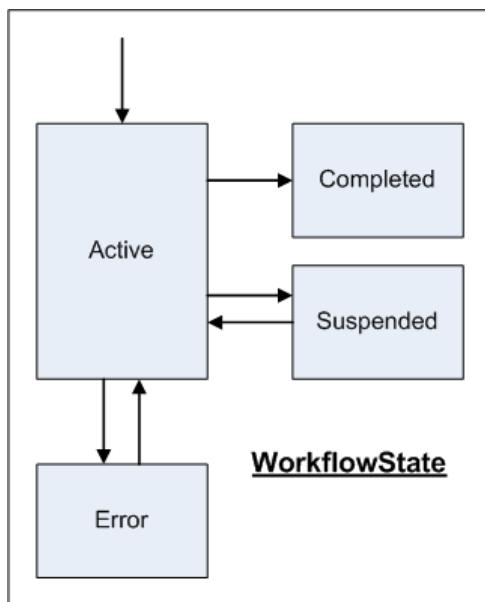
The newly created instance takes the form of a database entity called `Workflow`. (For more information on the `Workflow` entity, consult the BillingCenter *Data Dictionary*.) Because BillingCenter creates the `Workflow` entity in a bundle with other changes to its associated business data, BillingCenter does nothing with the workflow until it commits the workflow. BillingCenter 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 BillingCenter **Administration** interface, or from the command line, or through the Workflow API, the workflow moves to the `Suspended` state. A workflow in the `Suspended` state does nothing until manually resumed from the **Administration** interface, from the command line, or through the Workflow API.
- If an error occurs to a workflow executing in the background, the workflow moves into the `Error` state after it attempts the specified number of retries. A workflow in the `Error` state does nothing until manually resumed from the **Administration** interface, the command line, or the Workflow API.

The following graphic illustrates the possible workflow states:



Notice that this diagram does not convey any information about how an active workflow (a workflow in the Active state) is actually processing. For active workflows, Guidewire defines the workflow state in the `WorkflowActiveState` typelist, which contains the following states:

- `Running`
- `WaitManual`
- `WaitActivity`
- `WaitMessage`

Whether the workflow is actually running depends on whether it is the current *work item* being processed.

## Work Items

Each running workflow instance can have a *work item*. (See “Work Queues” on page 108 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` – BillingCenter retries a `Failed` work item up to the maximum retry limit.
- `Checkedout` – BillingCenter processes a `Checkedout` work item in a specific worker's queue after the work item reaches the head of that queue.

For the specifics of configuring work queues, see “Scheduling Work Queue Writers and Batch Processes” on page 113 in the *System Administration Guide*.

## Workflow Process Format

To structure a workflow script, Guidewire uses the concept of a directed graph that shows how the `Workflow` instance moves through the various states. (This is known formally as a Petri net or P/T net.) Guidewire calls each state a *Step* and calls a transition between two states a *Branch*. Guidewire defines multiple types of steps and branches.

Even though Guidewire defines the workflow scripts in XML files, you use Guidewire Studio to view, edit, manage, and create new workflows scripts. See “Workflow in Guidewire Studio” on page 336.

## 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	<b>Structural.</b> 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 “dlnqProcess” into the workflow text actually inserts “Workflow.DelinquencyProcess”.
<Start>	Exactly one	<b>Structural.</b> Element defining on which step the Workflow element starts. It can optionally contain Gosu code to set up the workflow or its business data.

Step	Script contains	Description
AutoStep ActivityStep ManualStep MessageStep	Zero, one, or more	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.  Each of these steps in turn can contain one or more of the following: <ul style="list-style-type: none"><li>• Any number of <code>Assert</code> code blocks for ensuring the conditions in the step are met.</li><li>• An <code>Enter</code> block with Gosu code to execute on entering the step.</li><li>• Any number of <code>Event</code> objects that generate on entering the step.</li><li>• Any number of <code>Notification</code> objects that generate on entering the step.</li><li>• An <code>Exit</code> block with Gosu code to execute on leaving a step.</li></ul>
		Several of these steps can contain other, step-specific, components: <ul style="list-style-type: none"><li>• An <code>ActivityStep</code> can contain any number of <code>Activity</code> steps that generate on entering the step.</li><li>• An <code>AutoStep</code> or <code>ActivityStep</code> can contain any number of <code>G0</code> branches which lead from this step to another step.</li><li>• A <code>ManualStep</code> can contain any number of <code>TRIGGER</code> 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 BillingCenter interface.)</li><li>• A <code>ManualStep</code> can contain any number of <code>TIMEOUT</code> branches that lead to another step after the elapse of a certain time.</li></ul>
Outcome	One or more	A specialized step that indicates a last step out of which no branch leaves.
<Finish>	Zero or one	<b>Structural.</b> An optional code block that contains Gosu code to perform any last cleanup after the workflow reaches an Outcome.

For more information on the `Workflows` editor, see “Using the Workflow Editor” on page 335.

## Workflow Gosu

Workflow elements `Start`, `Finish`, `Enter`, `Exit`, `G0`, `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 `G0`, `TRIGGER`, `TIMEOUT` elements as it encounters them upon following a branch.
- The Workflow engine runs `Finish` after it runs everything else.

Within the Gosu block, you can access the currently-executing workflow instance as `Workflow`. If you need to use local variables, declare them with `var` as usual in Gosu. However, if you need a value that persists from one step to another, create it as an extension field on `Workflow` and set its value from scripting. You can also create subflows in the Gosu blocks.

The current bundle for workflow actions is the bundle that the application uses to load the `Workflow` entity instance. The expression `Workflow.Bundle` returns the workflow bundle. See “Bundles and Database Transactions” on page 341 in the *Gosu Reference Guide*.

## Workflow Versioning

After you create a workflow script and make it active, it can create hundreds or even thousands of working instances in the BillingCenter 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 BillingCenter creates new instances of the workflow script, it uses the most recent script—the highest-numbered one—to run the workflow instance against.

It is possible to start a specific workflow with a specific version number. For details, see “Instantiating a Workflow” on page 363.

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. BillingCenter saves this number on the workflow instance as the `ProcessVersion` field.
- From then on, any time that the Workflow instance wakes up to execute, the Workflow engine uses the script with the same typecode and version number of the instance only.
- It is forbidden to have two workflow scripts with the same subtype and version number. The server refuses to start if you try.
- If a workflow instance cannot find a script with the right subtype and version number, it fails with an error and drops immediately into the `Error` state. (This might happen, perhaps, if someone inadvertently deleted the file or the file did not load for some reason.)

## When to Create a New Workflow Version

Guidewire recommends, as a general rule, that you create a new workflow version under most circumstances if you modify a workflow. For example:

- If you add a new step to the workflow, then create a new workflow version.
- If you remove an existing step from the workflow, then create a new workflow version.
- If you change the step type, for example, from Manual to an automatic step type, then create a new workflow version.

More specifically, for each workflow:

- BillingCenter records the current step of an active workflow in the database. Each change to the basic structure of a workflow requires a new version.
- BillingCenter records the branch that an active workflow selects in the database. A change to the Branch ID requires a new version.
- BillingCenter records the activity associated with an Activity step in the database. A change to an Activity definition requires a new version.
- BillingCenter records the trigger activity that occurs in an active workflow in the database. A removal of a trigger requires a new workflow version.
- BillingCenter records the `messageID` of each workflow message in the database. A modification to a `MessageStep` requires a new workflow version.

You do not need to create a new workflow version if you modify a constant such as the timeout value in the `TIMOUT` step. BillingCenter does record the wake-up time (for a `TIMOUT` step) that it calculates from the timeout time in the database. However, changing a timeout value does not affect workflows that are already on that step. Therefore, you do not need to create a new workflow version.

If you do modify a workflow, be aware that:

- If you convert a manual step to an automatic step, it can cause issues for an active workflow.
- If you reduce a timeout value, any active workflows that have already hit that step will only wait the previously calculated time.

---

**IMPORTANT** If there is an active workflow on a particular step, do not alter that step without versioning the workflow.

---

## Workflow Localization

At the start of the workflow execution, the Workflow engine evaluates the workflow locale and uses that locale for notes, documents, templates, and similar items. However, it is possible to set a workflow locale that is different from the default application locale through the workflow editor. This change then affects all notes, documents, templates, email messages, and similar items that the various workflow steps create or use.

You can also:

- Set a different locale for any spawned subworkflows.
- Set a locale for a Gosu block that a workflow executes.
- Set Studio to display a workflow step name in a different locale.

See “Localizing Guidewire Workflow” on page 71 in the *Globalization Guide* for details.

### To set a workflow locale

To view or modify the locale for a workflow, click in the background area of the layout view. This opens a properties area at the bottom of the screen. Enter a valid `ILocale` type in the `Locale` field to set the overall locale for a workflow. See “Localizing Guidewire Workflow” on page 71 in the *Globalization Guide* for details.

## Workflow Structural Elements

A workflow (or, more technically, a workflow XML script) contains a number of elements that perform a structural function in the workflow. For example, the `<Start>` element designates which workflow step actually initiates the workflow. Studio indicates the structural blocks by surrounding the block name with brackets in the workflow outline. (This reflects the XML-basis for these blocks.)

The workflow structural blocks include the following:

- `<Context>`
- `<Start>`
- `<Finish>`

### `<Context>`

Every workflow begins with a `<Context>` block. You use it to conveniently define symbols that apply to the workflow. You can use these symbols over and over in that workflow. For example, suppose that you extend the `Workflow` entity and add `User` as a foreign key. Then, you can define the symbol `user` for use in the workflow script with the value `Workflow.User`.

Within the workflow, you have access to additional symbols, basically whatever the workflow instance knows about. For example, you can define a symbol such that inserting `delinqProcess` into the workflow text actually inserts `Workflow.DelinquencyProcess`.

### 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:<sup>1</sup>

- Enter and Exit Scripts
- Asserts
- Events
- Notifications
- Branch IDs

The BillingCenter Administration tab displays the current step for each given workflow instance.

### Enter and Exit Scripts

A workflow step can have any amount of Gosu code in the Enter and Exit blocks to define what to do within that step. (Enter Script Gosu code is far more common.) To access the enter and exit scripts block, select a workflow step and view the properties tab at the bottom of the screen.

Enter Script	Gosu code that the Workflow engine runs just after it evaluates any Asserts (conditions) on the step. (That is, if none of the asserts evaluate to false. If this happens, the Workflow engine does not run this step.)
Exit Script	Gosu code that the Workflow engine runs as the final action on leaving this step.

For example, you could enter the following Gosu code for the enter script:

```
var msg = "Workflow " + Workflow.DisplayName + "started at " + Workflow.enteredStep
print(msg)
```

**Note:** If you rename a property or method, or change a method signature, and a workflow references that property or method in a Gosu field, BillingCenter throws `ParseResultsException`. This is the intended behavior. You must reload the workflow engine to correct the error (**Internal Tools** → **Reload** → **Reload Workflow Engine**).

## Asserts

A step can have any number of **Assert** condition statements. An **Assert** executes just before the **Enter** block. If an **Assert** fails, the Workflow engine throws an exception and handles it like any other kind of runtime exception. To access the **Assert** tab, select a workflow step.

<b>Condition</b>	Each condition must evaluate to a Boolean value.
<b>Error message</b>	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.

<b>Entity Name</b>	Entity on which to generate the event. This must a valid symbol name. See “<Context>” on page 346 for a discussion on how to use entity symbols in workflow Gosu.
<b>Event Name</b>	<p>Name of the event to generate. This must be a valid event name.</p> <ul style="list-style-type: none"> <li>• For general information on events, see “Messaging and Events” on page 303 in the <i>Integration Guide</i>.</li> <li>• For what constitutes a valid event name, specifically see “List of Messaging Events in BillingCenter” on page 323 in the <i>Integration Guide</i>.</li> </ul>

For example:

<b>Entity Name</b>	account
<b>Event Name</b>	someEvent

## Notifications

A step can have any number of non-blocking **Notification** activities. A notification in workflow terms is an activity that BillingCenter sends out, but which does not block the workflow from continuing. BillingCenter 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 351 for more information on activity generation.

<b>Name</b>	Name of the activity.
-------------	-----------------------

Pattern	Activity pattern code. This must be a valid activity pattern as defined through Guidewire BillingCenter.
Init	Optional Gosu code that the Workflow engine executes immediately after it creates the activity. Typically, you use this code to assign the activity. If you do not explicitly assign the activity, the Workflow engine auto-assigns the activity.

For example:

Name	notification
Pattern	generalReminder

## 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 BillingCenter 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 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	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 347 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), BillingCenter stores an optional response string (supplied with the ack) on the message in the **Response** field. BillingCenter 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.

Field	Description
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

- 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 347 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. BillingCenter indicates the termination of an activity by marking it one of the following:

- Completed (which includes either being approved or rejected)
- Skipped
- Canceled

Activities are a convenient way to send messages and questions asynchronously to users who might not even be logged into the application.

The **Workflows** editor indicates an activity step with a person icon in the box the represents that step.



Within an **ActivityStep**, you specify one or more activities. The Workflow engine creates each defined activity as it enters the step. (This occurs immediately after the Workflow engine executes the **Enter Script** block, if there is one.) The activity is available on all steps.

The only difference between an **Activity** and a **Notification** within a workflow is that:

- An **Activity** pauses the workflow until all the activities in the step terminate.
- A **Notification** does not block the workflow from continuing.

If more than one **Activity** exists on an **ActivityStep**, then the Workflow engine generates all of them immediately after the **Enter** block (along with any events or notifications). The step then waits for all of the activities to terminate. If desired, an **ActivityStep** can also contain TIMEOUT and TRIGGER branches as well. In that case, if a timeout or a trigger on the step occurs, then the workflow does not wait for all the activities to complete before leaving the step.

After BillingCenter 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 BillingCenter 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 BillingCenter.
ID	ID of a branch leaving this step. It defaults to the <b>To</b> 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 BillingCenter. To view a list of valid activity pattern codes, view the <b>ActivityPattern</b> typelist. Only enter a value in the <b>Pattern</b> field that appears on this typelist. For example: <ul style="list-style-type: none"> <li>• approval</li> <li>• approvaldenied</li> <li>• general</li> <li>• ...</li> </ul>
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> <p>The initialization code creates an activity based on the activity pattern that you set in the <b>Pattern</b> field.</p>

## ManualStep

A **ManualStep** is a step that waits for an external TRIGGER to be invoked or a TIMEOUT to pass. Unlike **AutoStep** or **ActivityStep**, a **ManualStep** must not have, and cannot have, GO branches leaving it. However, it can have zero or more TRIGGER branches or zero, or more, TIMEOUT branches. It must have at least one of these branches. Otherwise, there would be no way to leave this step.

The **Workflows** editor indicates a manual step with an hour-glass icon in the box the represents that step.



### Manual Step with Timeout

If you specify a *timeout* for this step, then you also need to specify one of the following. (See also “TIMEOUT” on page 358 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"> <li>Trigger: A valid trigger key as defined in typelist <code>WorkflowTriggerKey</code>.</li> <li>Timeout: ID of a branch leaving this step. It defaults to the To value if you do not supply a value.</li> </ul>
To	ID of the step to which the workflow goes if the condition specified for this branch evaluates to true.
Time Delta	Specifies a fixed amount of time to pause before continuing. For example, the following sets the wait time to 60 minutes (one hour): 3600s,
Time Absolute	Specifies a fixed point in time. For example, the following sets the point to continue to after the policy <code>CancelProcessDate</code> : <code>PolicyPeriod.Cancellation.CancelProcessDate</code>

If the `WorkflowTriggerKey` typelist does not contain any trigger keys, then you do not see the Trigger option in the dialog.

3. Click on your newly created step. It is possible that there are additional tabs to fill out in the properties area at the bottom of the screen.

### Outcome

An **Outcome** is a special step that has no branches leading out of it. It is thus a final or terminal step. If a workflow enters any **Outcome** step, it is complete. It is possible (and likely) for a workflow to have multiple outcomes or final steps.

The **Workflows** editor indicates an outcome step with a gray bar in the box to indicate that this is a final step.

Outcome

After the Workflow engine successfully enters an **Outcome** step (meaning that the Workflow engine successfully executes the **Enter** block of the **Outcome** step), it does the following:

1. The workflow generates all the listed events and notifications.
2. It executes the **<Finish>** block of the workflow process.
3. It changes the state of the workflow instance to **Completed**.

You must structure each workflow script so that its execution eventually and inevitably leads to an **Outcome**. Otherwise, you risk infinitely-running workflows, which means that the load on the Workflow engine can increase linearly over time, crippling performance.

#### To create a new outcome step

1. Right-click in the workflow workspace, and select **New Outcome**.
2. Enter a step ID in the **New Outcome** dialog.
3. Click on your newly created step. It is possible that there are additional tabs to fill out in the properties area at the bottom of the screen.

## Step Branches

A branch is a transition from one step to another. There are multiple kinds of elements that facilitate branching to another step. They are:

- GO
- TRIGGER
- TIMEOUT

The **Workflows** editor indicates a branch by linking two steps with a line and placing one of the following icons on the line to indicate the branch type.

Type	Icon	Description
GO		A branch or transition to another workflow step. It occurs only within an <b>AutoStep</b> workflow step. <ul style="list-style-type: none"> <li>• If there is only a single GO branch within the workflow step, branching occurs immediately upon workflow reaching that point.</li> <li>• 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.</li> </ul>
TRIGGER		A branch or transition to another workflow element. It occurs only within a <b>ManualStep</b> 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 <b>ManualStep</b> workflow step. Branching to another workflow step occurs only after the passing of a specific time interval.

All branch elements contain a **To** value that indicates the step to which this branch leads. It can also contain an optional embedded Gosu block for the Workflow engine to execute if a workflow instance follows that branch.

How a workflow decides which branch to take depends entirely on the type of the branch. However, the order is always the same:

- The Workflow engine executes the **Enter** block for a given step and generates any events, notifications, and activities (waiting for these activities to complete).

- The Workflow engine attempts to find the first branch that is ready to be taken. It starts with the first branch listed for that step in the outline view, then moves to evaluate the next branch if the previous branch is not ready.
- If no branch is ready (which is possible only on a `ManualStep`), the workflow waits for one to become ready.
- After the Workflow engine selects a branch, it runs the `Exit` block, then executes the Gosu block of the branch.
- Finally, the workflow moves to the next step and begins to evaluate it.

## Working with Branch IDs

Every branch also has an ID, which is its reference name. An ID is necessary because the Workflow instance sometimes needs to persist to the database which branch it is trying to execute. (This can happen, for example, if an error occurs in the branch and the workflow drops into the `Error` state). A branch ID must be unique within a given step.

If you do not specify an ID for a branch (which occurs frequently), the workflow uses the value of `nextStep` attribute as a default. This works well except in the special case in which you have more than one branch leading from the same `Step A` to the same `Step B`. (This can happen, for example, if you want to OR multiple conditions together, or if you want different Gosu in the different branches but the same `nextStep`.) In that case, you must add an ID to each of those branches. Studio complains with a verification error upon loading (or reloading) the workflow scripts if you do not do this.

Do the following to assign an ID to each type of branch:

Type	Action to take
GO	Optionally add an ID to a GO branch. If you do not provide one, Studio defaults the ID to the value of the <code>nextStep</code> attribute. However, Guidewire recommends that you create specific IDs if there are multiple GO branches that all move to the same next step.
TRIGGER	Always add an ID to a TRIGGER branch. Guidewire requires this as you must invoke a trigger explicitly. You must use a value from the <code>WorkflowTriggerKey</code> typelist for the branch ID.
TIMEOUT	Optionally add an ID to a TIMEOUT branch. If you do not provide one, Studio defaults the ID to the value of the <code>nextStep</code> attribute.

## GO

The simplest kind of branch is `GO`. It appears on `AutoStep`, `ActivityStep` and `MessageStep`. There can be a single `GO` branch or a list of multiple `GO` branches. If there is a single `GO` branch, then you need only specify the `To` field and any optional Gosu code. The Workflow engine takes this `GO` branch immediately as it checks its branches.

The `Workflows` editor indicates a `GO` branch with an arrow icon superimposed on the line that links the two steps. (That is, the initial `From` step and the `To` step to which the workflow goes if the `GO` condition evaluates to `true`.)

To access the dialog that defines the `GO` branch, right-click the starting step—in this case, `CheckOnOrder`—and select `New GO` from the menu. (Studio only displays those choices that are appropriate for that step.) This dialog contains the following fields:

Field	Description
Branch ID	ID of the branch to create
From	ID of the step on which the <code>GO</code> branch starts.
To	ID of the step on which the <code>GO</code> branch starts.

As discussed (in “Working with Branch IDs” on page 355), 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 BillingCenter interface or from a Gosu call.) Guidewire requires a branch ID field on all TRIGGER elements, as outside code uses the ID to manually reference the branch.

Unlike all other the IDs used in workflows, TRIGGER IDs are not plain strings but typelist values from the extendable `WorkflowTriggerKey` typelist. This provides necessary type safety, as scripting invokes triggers by ID. However, it also means that you must add new typecodes to the typelist if you create new trigger IDs.

### Invoking a Trigger

How does one actually invoke a TRIGGER? Almost anything can do so, from Gosu rules and classes to the BillingCenter interface. Typically, in BillingCenter, 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 366 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 BillingCenter *Data Dictionary* entry on `Workflow`.)

The Workflow engine then *wakes up* the workflow instance and the TRIGGER inspects the `triggerInvoked` field to see if something invoked the trigger. Depending on how you set the `invokeTrigger` method parameters, the Workflow engine handles the result of the TRIGGER either synchronously or asynchronously.

### Creating a Trigger Branch

To access the TRIGGER branch dialog, right-click the starting step and select **New Trigger** from the menu. (Studio only displays those choices that are appropriate for that step.) This dialog contains the following fields:

Field	Description
Branch ID	Name of this branch as defined in the <code>WorkflowTriggerKey</code> type list. Select from the drop-down list.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.

After you create the branch, click on the link (line) that runs between the two steps. You see the following fields, which are identical to those used to define a GO branch:

Field	Description
Branch ID	Automatically generated.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.
Arrow Visible	Show an arrow head on the branch line to indicate direction.
Description	Description of this branch.
Condition	Must evaluate to either true or false.
Execution	Gosu code to execute if the Workflow engine takes this branch.

### Trigger Availability

Simply because you define a TRIGGER on a `ManualStep` does not mean it is necessarily available. You can restrict trigger availability in the following different ways:

- You can specify user access permission through the use of the `Permission` field.
- You can add any number of `Available` conditions on the `Available` tab to further restrict availability. If the condition expression evaluates to `true`, the trigger is available. Otherwise, it is unavailable.

For example (from PolicyCenter), the following Gosu code indicates that the workflow can only take this branch if a user has permission to rescind a policy. (The condition evaluates to `true`.)

```
PolicyPeriod.CancellationProcess.canRescind().Okay
```

## TIMEOUT

Another kind of branch is TIMEOUT, which (like TRIGGER) can appear on ManualStep or an ActivityStep. You still have a To field and optional Gosu block. However, instead of using a condition to determine how to move forward, the Workflow engine executes the TIMEOUT element after the elapse of a specified amount of time.

You can use a TIMEOUT in the following ways:

- As the default behavior for a stalled workflow. For example:

*Do x if BillingCenter has not invoked a trigger for a certain amount of time.*

- As a deliberate delay. For example:

*Go to sleep for 35 days.*

You can specify the time to wait using one of the following attributes. (Studio complains if you use neither or both.)

- timeDelta
- timeAbsolute.

### The Time Delta Value

The Time Delta value specifies an amount of time to wait, starting from the time the Workflow instance successfully enters the step. (The wait time starts immediately after the Workflow engine executes the Enter Script block for the step.) You specific the time to wait with a number and a unit, for example:

- 100s for 100 seconds
- 15m for 15 minutes
- 35d for 35 day

You can also combine numbers and units, for example, 2d12h30m for 2 days, 12 hours, and 30 minutes.

### The Time Absolute Value

Often, you do not want to wait a certain amount of time. Instead, you want the step to time out after passing a certain point relative to a date in the business model (for example, five days after a specific event occurs). In that case you can set the Time Absolute value, which is a Gosu expression that must resolve to a date.

**IMPORTANT** Do not use the current time in a Time Absolute expression. The Workflow engine re-evaluates this expression each time it checks TIMEOUT. For example, the time-out never ends for the following expression, `java.util.Date.CurrentDate + 1`, as the expression always evaluates to the future.

### Creating a Timeout Branch

The following graphic illustrate how you define a Timeout branch in the Workflows editor. To access the Timeout branch dialog, right-click the starting step and select New Timeout from the menu. Notice that you must enter either time absolute expression or a time delta value. This dialog contains the following fields:

Field	Description
Branch ID	Name you choose for this branch.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.
Time Delta	Time to wait, starting from the time the Workflow instance successfully enters the step.
Time Absolute	Gosu expression that must resolve to a fixed date.

After you create the branch, click on the link that runs between the two steps. You see the following fields:

Field	Description
Branch ID	Automatically generated.
From	Automatically generated. Workflow step ID of the beginning point of the branch.
To	Workflow step ID of the ending point of the branch.
Arrow Visible	Show an arrow head on the branch line to indicate direction.
Time Delta	Time to wait, starting from the time the Workflow instance successfully enters the step.
Time Absolute	Gosu expression that must resolve to a fixed date.
Execution	Gosu code to execute if the Workflow engine takes this branch.

## Creating New Workflows

To create a new workflow, you can do the following:

Action	Description
Cloning an Existing Workflow	Creates an exact copy of an existing workflow type, with the same name but with an incremented version number. (This process clones the workflow with highest version number, if there multiple versions already exist.) Perform this procedure if you merely want a new version of an existing workflow.
Extending an Existing Workflow	Creates a new (blank) workflow with a name of your choice based on the workflow type of your choice.

### Cloning an Existing Workflow

Cloning an existing workflow is a relatively simple process. Also, if you clone an existing, fully built workflow, then you can leverage the work of the original workflow. However, you can only clone existing workflow types. You cannot use this method to create a new workflow type.

#### To clone an existing workflow

1. Open the **Workflows** node in the Project window tree.
2. Select an existing workflow type, right-click and select **New → Workflow** from the menu.

Studio creates a cloned, editable copy of the workflow process and inserts it under the workflow node with an incremented version number. You can then modify this version of the workflow process to meet your business needs.

### Extending an Existing Workflow

To extend an existing workflow, you must create an **.eti** (extension) file and populate it correctly. To assist you, Studio provides a dialog in which you can enter the basic workflow information. You must then enter this information in the **.eti** file.

#### To extend an existing workflow

1. First, determine the workflow type that you want to extend.
2. Select **Workflows** in the Project window, right-click and select **Create metadata for a new workflow subtype** from the menu.

3. In the **New Workflow subtype metadata** dialog, enter the following:

Field	Description
Entity	The workflow object to create.
Supertype	The type or workflow to extend. You can always extend the <b>Workflow</b> type, from which all subtypes extend.
Description	Optional description of the workflow.
Foreign keys	Click the <b>Add</b> button to enter any foreign keys that apply to this workflow object.

4. Click **Gen to clipboard**. This action generates the workflow metadata information in the correct format and stores on the clipboard.
5. Expand the **Extensions** folder in the **Project** window.
6. Right-click the **Entity** folder and select **New → Entity** from the menu.
7. Enter the name of the file to create in the **New File** dialog. Enter the same value that you entered in the **New Workflow subtype metadata** dialog for **Entity** and add the **.eti** extension. Studio then creates a new `<entity>.eti` file. Open this file, right-click, and choose **Paste** from the menu. Studio pastes in the metadata workflow that you created in a previous step. For example, if you extend **Workflow** and create a new workflow named **NewWorkflow**, then you must create a new **NewWorkflow.eti** file that contains the following:
- ```
<?xml version="1.0"?>
<subtype desc="" entity="NewWorkflow" supertype="Workflow"/>
```
8. (Optional) To provide the ability to localize the new workflow, add the following line of code to this file (as part of the **subtype** element):
- ```
<typekey desc="Language" name="Language" typelist="LanguageType"/>
```
- Continuing the previous example, you now see the following:
- ```
<?xml version="1.0"?>
<subtype desc="" entity="NewWorkflow" supertype="Workflow">
  <typekey desc="Language" name="Language" typelist="LanguageType"/>
</subtype>
```
9. Stop and restart Guidewire Studio so that it picks up your changes.
- You now see **NewWorkflow** listed in the **Workflow** typelist.
  - You now see an **NewWorkflow** node under **Resources → Workflows**.
10. Select the **NewWorkflow** node under **Workflows**, right-click and select **New Workflow Process** from the menu. Studio opens an empty workflow process that you can modify to meet your business needs.

## Extending a Workflow: A Simple Example

This simple examples illustrates the following steps:

- Step 1: Extend an Existing Workflow Object
- Step 2: Create a New Workflow Process
- Step 3: Populate Your Workflow with Steps and Branches

### Step 1: Extend an Existing Workflow Object

To extend an existing workflow object, review the steps outlined in “Extending an Existing Workflow” on page 359. For this example, you create a new **ExampleWorkflow** object by extending (subtyping) the base **Workflow** entity.

**To extend a workflow object**

1. Create a new ExampleWorkflow.eti file and enter the following:

```
<?xml version="1.0"?>
<subtype desc="" entity="ExampleWorkflow" supertype="Workflow">
  <typekey desc="Language" name="Language" typelist="LanguageType"/>
</subtype>
```

2. Close and restart Studio.

You now see an ExampleWorkflow entry added to the Workflow typelist and a new ExampleWorkflow workflow type added to Workflows in the Resources tree.

**Step 2: Create a New Workflow Process**

Next, you need to create a new workflow process from your new ExampleWorkflow type.

**To create a new workflow process**

1. Select ExampleWorkflow from Workflows in the Project window.
2. Right-click and select New Workflow from the menu.

Studio opens an outline view and layout view for the new workflow process:

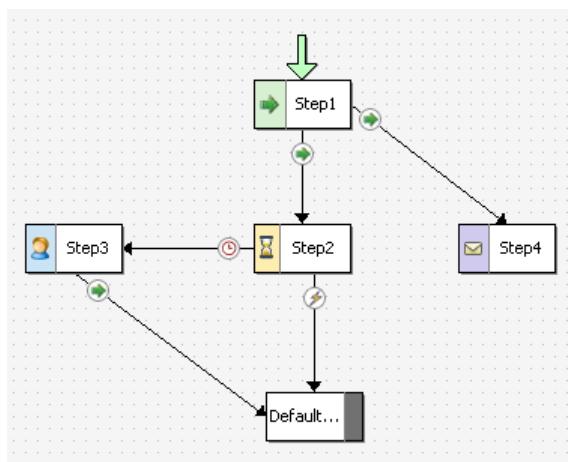
- The outline view contains the few required workflow elements.
- The layout view contains a default outcome (DefaultOutcome).

**Step 3: Populate Your Workflow with Steps and Branches**

Finally, to be useful, you need to add outcomes, steps, and branches to your workflow. This examples creates the following:

- A Step1 (AutoStep) with a default GO branch to the DefaultOutcome step, which you designate as the first step in the <Start> element
- A Step2 (ManualStep) with a TRIGGER branch to the DefaultOutcome step
- A Step3 (ActivityStep) with a GO branch to the DefaultOutcome step
- A TIMEOUT branch from Step2 to Step3, with a 5d time delta set
- A Step4 (MessageStep) with a GO branch from Step1 to Step4

The example workflow looks similar to the following:



This example does not actually perform any function. It simply illustrates how to work with the dialogs of the Workflows editor.

### To add steps and branches to a workflow

1. Right-click within an empty area in the layout view and select **New AutoStep** from the menu:
  - For **Step ID**, enter Step1.
  - Do not enter anything for the other fields.Studio adds your autostep to the layout view and connects Step1 to DefaultOutcome with a default GO branch.
2. Select <Start> in the outline view (middle pane):
  - Open the **First Step** drop-down in the property area at the bottom of the screen.
  - Select Step1 from the list. This sets the initial workflow step to Step1.
  - Save your work.
3. Right-click within an empty area in the layout view and select **New ManualStep** from the menu:
  - For **Step ID**, enter Step2.
  - For branch **Type**, select TRIGGER.
  - For trigger **ID**, select Cancel.The ID value sets a valid trigger key as defined in typelist WorkflowTriggerKey. If Cancel does not exist, then choose another trigger key. If no trigger keys exist in WorkflowTriggerKey, then you must create one before you can select TRIGGER as the type.
4. Select the GO branch (the line) leaving Step1:
  - In the property area at the bottom of the screen, change the **To** field from DefaultOutcome to Step2. Studio moves the branch to link the specified steps.
  - Realign the steps for more symmetry, if you choose.
5. Right-click within an empty area in the layout view and select **New ActivityStep** from the menu:
  - For **Step ID**, enter Step3.
  - For **Name**, enter ActivityPatternName.
  - For **Pattern**, enter NewActivityPattern.
6. Select Step3, right-click, and select **New TIMEOUT** from the menu:
  - For **Branch ID**, enter TimeoutBranch.
  - For **Time Delta**, enter 5d. This sets the absolute time to wait to five days.
  - For **To**, select Step3.Studio adds a branch from Step2 to Step3 and adds the timeout symbol to it.
7. Right-click within an empty area in the layout view and select **New MessageStep** from the menu:
  - For **Step ID**, enter Step4.
  - For **Dest ID**, enter 89 (or any valid message destination ID).
  - For **Event Name**, enter EventName.Studio adds the step to the layout view and creates a link between Step4 and DefaultOutcome.
8. Select the new link from Step4 to DefaultOutcome.
  - In the property area at the bottom of the screen, change **Arrow Visible** to **false** to delete this link.Studio removes the link (branch).
9. Select Step1, right-click, and select **New GO** from the menu:
  - For **Branch ID**, enter Step4.
  - For **To**, select Step4.Studio adds the new GO branch between Step1 and Step4.

## Instantiating a Workflow

It is not sufficient to create a workflow. Generally, you want to do something moderately useful with it. To perform work, you must instantiate your workflow and call it somehow.

Suppose, for example, that you create a new workflow and call it, for lack of a better name, `HelloWorld1`. You can then instantiate your workflow using the following Gosu:

```
var workflow = new HelloWorld1()
workflow.start()
```

### Starting a Workflow

There are multiple workflow `start` methods. The following list describes them.

<code>start()</code>	Starts the workflow.
<code>start(version)</code>	Starts the workflow with the specified process version.
<code>startAsynchronously()</code>	Starts the workflow asynchronously.
<code>startAsynchronously(version)</code>	Starts the workflow with the specified process version asynchronously.

For information on versioning works with workflow, see “Workflow Versioning” on page 344.

### 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 BillingCenter <code>Workflow</code> console

### See Also

- See Workflow Debugging, Logging, and Testing for more information.

## A Simple Example of Instantiation

The following example creates a trivial workflow named `HelloWorld1`. The objective of this example is not to show the branching structure that you can create in workflow. Rather, the purpose of this exercise is to construct the workflow, trigger the workflow, and examine the workflow in the BillingCenter `Workflow` console. The example keeps the workflow as simple as possible. The workflow consists of the following components:

- `<Context>`
- `<Start>`
- `Step1`
- `Step2`
- `DefaultOutcome`
- `<Finish>`

### A Simple ClaimCenter Example

**Note:** This example uses business entities and rules that apply specifically to the Guidewire ClaimCenter application. However, the particular business objects are not important. What is more important is how you create and instantiate a workflow process.

For the workflow to run and do some work and appear on the workflow console, the example instantiates it from a Claim Update rule. If you attempt to instantiate the workflow from a link or button on a Claim view screen (`Claim Summary`, for example) the workflow executes but does not update anything. Also, it does not appear in the `Workflow` console.

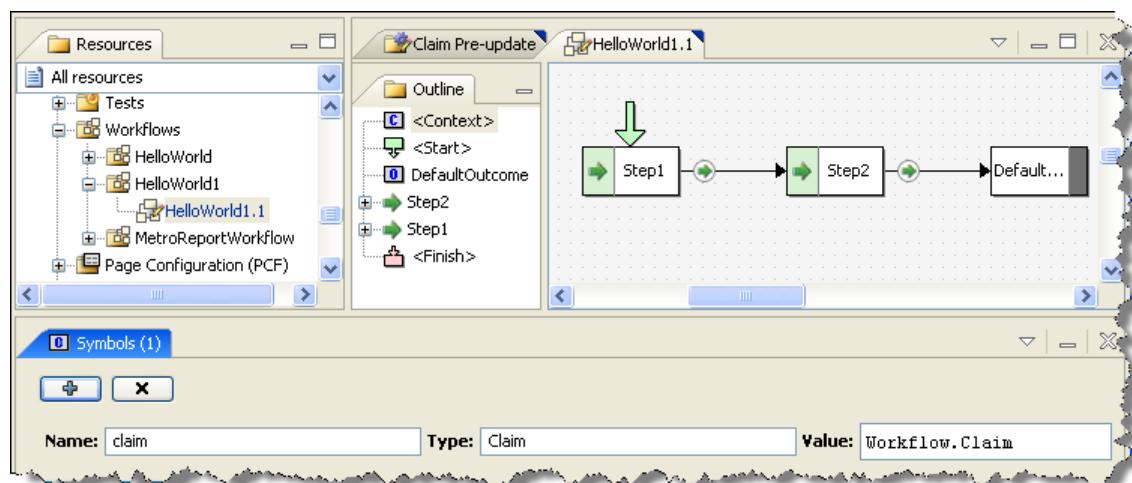
To cause updates to happen, the example instantiates the workflow from an **Edit** screen in ClaimCenter. It then calls a Claim Pre-Update Rule in Studio.

### To create a simple workflow and instantiate it

1. Create a `HelloWorld1.eti` file (in Extensions → Entity) and populate it with the following:

```
<?xml version="1.0"?>
<subtype desc="HelloWorld 1 Example Workflow"
    entity="HelloWorld1"
    supertype="ClaimWorkflow">
    <typekey desc="Language" name="Language" typelist="LanguageType"/>
</subtype>
```

2. Stop and restart Studio.
3. Select your new workflow type from the **Workflows** node. Right-click and select **New → Workflow Process**.
4. Create a simple workflow process similar to the following. It does not need to be complex, as it simply illustrates how to start a workflow from the ClaimCenter interface.



Notice that it has a `claim` symbol set in `<Context>`.

5. For Step1, add the following to the Enter block for that step:

```
gw.api.util.Logger.logInfo( "HelloWorld1 step 1, step called ClaimNumber " + claim.ClaimNumber)
Workflow.log( "HelloWorld Step 1", "HelloWorld1 step 1 entered: Claim Number " + claim.ClaimNumber )
```

6. For Step2, add the following to the Enter block for that step:

```
gw.api.util.Logger.logInfo( "HelloWorld1 step 2, step called ClaimNumber " + claim.ClaimNumber)
Workflow.log( "HelloWorld Step 2", "HelloWorld1 step 2 entered: Claim Number " + claim.ClaimNumber )
```

7. Create a simple Claim Pre-Update rule similar to the following:

- The *rule condition* specifies that the Workflow engine instantiates the workflow only if the claim `PermissionRequired` property is set to `fraudriskclaim`.
- The *rule action* instantiates the `HelloWorld1` workflow. It first tests for an existing `HelloWorld1` workflow that is not in the completed state and that has the same claim number as the one being updated. If it does not find a matching workflow, then the Workflow engine instantiates `HelloWorld1` and logs the information.

#### Rule Conditions:

```
claim.PermissionRequired=="fraudriskclaim"
```

#### Rule Actions:

```
gw.api.util.Logger.logInfo( "Entering Pre-Update" )

var hw_wf = claim.Workflows.firstWhere( \ c -> c.Subtype == "HelloWorld1"
    && (c as entity.HelloWorld1).State != "completed"
    && (c as entity.HelloWorld1).Claim.ClaimNumber==claim.ClaimNumber)
```

```
if (hw_wf == null) {  
    gw.api.util.Logger.logInfo( "# Studio instantiating HelloWorld1 and starting it!" )  
    var workflow = new entity.HelloWorld1()  
    workflow.Claim = claim  
    workflow.start()  
}
```

8. Log into ClaimCenter and open any sample claim.
9. Navigate to the **Claim Summary** page, then select the **Claim Status** tab.
10. Click **Edit** and set the **Special Claim Permission** value to **Fraud risk**.
11. Click **Update**. This action triggers the `HelloWorld1` workflow.

#### To view the server console

1. Navigate to the application server console.
2. View the logger statements.

#### To view the Workflow console

1. Log into ClaimCenter using an administrative account.
2. Navigate to the **Administration** tab and select **Workflows** from the left-side menu.
3. Click **Search** in the **Find Workflows** screen. You do not need to enter any search information. Studio displays a list of workflows, including `HelloWorld1`.
4. Select `HelloWorld1` from the list and view its details.

## The Workflow Engine

The Workflow engine is responsible for processing a workflow. It does this by looking up and executing the appropriate Workflow Process Script. This script (often just called Workflow Process or Workflow Script) is an XML file that the Studio Workflow editor generates, and which you manage in Studio. The base configuration workflow scripts live in the `modules/config/workflow` directory.

### Distributed Execution

BillingCenter uses a work queue to handle workflow execution. This, in simple terms, means that you can have a whole cluster of machines that:

- Wake up internal `Workflow` instances,
- Advance them as far as they can go,
- Then, let them go back to sleep if they need to wait on a timeout or activity.

Asynchronous workflow execution always works the same way:

1. BillingCenter creates a `WorkflowWorkItem` instance to advance the workflow.
2. The worker instance picks up the work item.
3. The work item retrieves the workflow and advances it as far as possible (to a `ManualStep` or `Outcome`).

You can create a work item in any of the following different ways:

- By a call to the `AbstractWorkflow.startAsynchronously` method
- By invoking a trigger with `asynchronous = true`
- By completing a workflow-linked activity
- By the `Workflow` batch process, which queries for active workflows waiting on an expired timeout

- By a call to `AbstractWorkflow.resume`, typically initiated by an administrator using the workflow management tool

After the workflow advances as far as it can, BillingCenter 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 363. 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 BillingCenter interface (a user clicking a button) or from Gosu. In either case, this is done through a call to the `invokeTrigger` method on a `Workflow` instance.

The action of completing an activity or the receipt of a message response automatically creates a work item to advance the workflow. A background batch process checks for timeout elements. It is responsible for finding timed-out workflows that are ready to advance and creating a work item to advance them.

### The `invokeTrigger` Method

If a user (or Gosu code) invokes an available trigger (`TRIGGER`) on a `ManualStep`, the workflow can execute either synchronously or asynchronously. A Boolean parameter in the `invokeTrigger` method determines the execution type. This method takes the following signature:

```
void invokeTrigger(WorkflowTrigger triggerKey, boolean synchronous)
```

For example (from PolicyCenter):

```
policyPeriod.ActiveWorkflow.invokeTrigger( trigger, false )
```

The `trigger` parameter defines the TRIGGER to use. This must be a valid trigger defined in the `WorkflowTriggerKey` type list.

The `synchronous` value in this method has the following meanings:

<code>true</code>	(Default) Instructs the workflow to immediately execute in the current transaction and to block the calling code until the workflow encounters a new stopping point.
<code>false</code>	Instructs the workflow to run in the background, with the calling code continuing to execute. The workflow continues until it encounters a new stopping point.

### Trigger Availability

For a trigger to be available, the workflow execution sequence must select a branch for which both of the following conditions are true:

- A trigger must exist on the step.
- There is no other determinable path (which usually means that no timeout has already expired).

Thus, if both of these conditions are true, after an invocation to the `invokeTrigger` method, the Workflow engine starts to advance the workflow from the selected branch again.

### Invoking a Trigger

Invoking a trigger (either synchronously or asynchronously) does the following:

1. It updates the workflow. Any changes made to a transaction bundle that were committed by the actual invocation of the trigger, are committed.
2. It causes the workflow to create a log entry of the trigger request. If there is an error in the workflow advance, any request to the workflow to resume causes the process to start again. (See also “Workflow Administration” on page 369.)
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 BillingCenter uses for display keys, dates, numbers, and other similar items. By default, this is the application default locale. It is important for the Workflow engine to determine the locale as it is possible to override this locale for any specific workflow subtype. You can also override the locale in the workflow definition on the workflow element. See “Localizing Guidewire Workflow” on page 71 in the *Globalization Guide* for more information.
  - b. It steps through each of the workflow steps (meaning that it performs all the actions within that step) until it cannot keep going.
  - c. It commits the transaction associated with the executed steps to the database.

### Error Handling and Transaction Rollback

If there is an error during a workflow step, the Workflow engine rolls the database back and leaves it in the state that it was. If working with an external system, you need to one of the following:

- You need to design the services in the external system, or,
- You need to use the Guidewire message subsystem to keep an external system state in synchronization with the application database state.

It is important to understand whether a workflow executes synchronously or asynchronously as it affects errors and transaction rollbacks:

Execution type	Application behavior
Synchronous	If any exception occurs during <i>synchronous</i> execution, even after the workflow has gone through several steps, BillingCenter rolls back all workflow steps (along with everything else in the bundle). The error cascades all the way up to the calling code (the code that started the workflow or invoked the trigger on the workflow). <ul style="list-style-type: none"> <li>• If you start the workflow or invoke the trigger from the BillingCenter interface, BillingCenter displays the exception in the interface.</li> <li>• If some other code started the workflow, that code receives the exception.</li> </ul>
Asynchronous	If any exception occurs during <i>asynchronous</i> execution (as it executes in the background), BillingCenter logs the exception and rolls back the bundle, in a similar manner to the synchronous case. <p>BillingCenter then handles workflow retries in the standard way through the worker. BillingCenter 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, BillingCenter marks the work item as failed and it puts the workflow into the <code>Error</code> state. A workflow in the <code>Error</code> state merely sits idle until you restore it from the <b>Administration</b> tab within BillingCenter. Restoring the workflow creates another work item.</p> <p>After you manually restore a workflow from an <code>Error</code> to an <code>Active</code> state, it again tries to resume whatever it was doing as it left off, typically:</p> <ul style="list-style-type: none"> <li>• entering the step</li> <li>• following the branch</li> <li>• or, attempting to perform whatever it was doing at the time the exception occurred</li> </ul> <p>Of course, if you have not corrected the problem that caused the error, then the workflow can drop right back into <code>Error</code> state again. This is only after the work item performs its specified number of retries, however.</p>

## Guidelines

In practice, Guidewire recommends that you keep the following guidelines in mind as you work with workflows:

- If you invoke a workflow `TRIGGER`, do so synchronously if you need to make immediate use (in code) of the results of that trigger. For this reason, the BillingCenter rendering framework typically always invokes the trigger synchronously. But notice that you only get immediate results from an `AutoStep` that might have executed. If the workflow encounters a `ManualStep` or an `ActivityStep`, it immediately goes into the background.
- If you complete an activity, it does not synchronously (meaning immediately) advance the workflow. Instead, a background process checks for workflows whose activities are complete and which are therefore ready to move forward. Guidewire provides this behavior, as otherwise, if an error occurs, the user who completes the activity sees the error, which is possibly confusing for that user.
- If you invoke a workflow `TRIGGER` from code that does not necessarily care whether there was a failure in the workflow, you need to invoke the `TRIGGER` asynchronously. (You do this by setting the `synchronous` value in the workflow method to `false`.) That way, the workflow advances in the background and any errors it encounters force the workflow into the `Error` state. The exception does not affect the caller code. However, the calling code creates an exception if it tries to invoke an unavailable or non-existent workflow `TRIGGER`. Messaging plugins, in particular, need to always invoke triggers asynchronously.

## Workflow Subflows

A workflow can easily create another child workflow in Gosu using the scriptable `createSubFlow` method on `Workflow`. There are multiple versions of this method:

```
Workflow createSubFlow(workflow)
Workflow createSubFlow(workflow, version)
```

A subflow has the same foreign keys to business data as the parent flow. It also has an edge foreign key reference to the caller `Workflow` instance, appropriately accessed as `Workflow.caller`. (If internal code, and not some other workflow, calls a *macro* workflow, this field is `null`.)

Each workflow also has a `subFlows` array that lists all the flows created by the workflow, including the completed ones. (This array is empty for workflows that have yet to create any subflows.) The Gosu to access this array is:

```
Workflow.SubFlows
```

You can use subflows to implement simple parallelism in internal workflows, which is otherwise impossible as a single workflow instance cannot be in two steps simultaneously. For example, it is possible for the macro flow to create a subflow in step A. It can then leave this subflow to do its own work, and only wait for it to complete in step E. It is your responsibility as the one configuring the macro workflow to decide how to react if a subflow drops into `Error` mode or becomes canceled for some reason.

#### See also

- “Creating a Locale-Specific Workflow SubFlow” on page 74 in the *Globalization Guide*

## Workflow Administration

You can administer workflow in any of the following ways:

- Through the BillingCenter **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 BillingCenter **Administration** interface is error handling. Errors can be the following:

- A few workflows fail
- Or, in a worst case scenario, thousands fail simultaneously

Finding workflows that have not failed but have been idling for an extremely long time is also likely. A secondary use is just looking at all the current running flows to see how they work. Guidewire therefore organizes the **Administration** interface for workflow around a search screen for searching for workflow instances. You can filter the search screen, for example, by instance type, state (especially `Error` state), work item, last modified time, and similar criteria.

A user with administrative permissions can search for workflows from the **Administration** → **Workflows** page. However, to actually manage workflow, that user must have the `workflowmanage` permission. In the base BillingCenter 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, BillingCenter 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, BillingCenter 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, BillingCenter suspends the selected workflows. This action is appropriate for all workflow and steps. However, BillingCenter 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, BillingCenter 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

BillingCenter collects workflow statistics periodically and captures the elapse and execution time for individual workflow types and steps. You can search by workflow type and date range.

### Workflow and Server Tools

Those with access to the Server Tools, can also access the following:

Batch Process Info	Use to view information on the last run-time of a writer, and to see the schedule for its next run-time. From this page, you also have the ability to stop and start the scheduling of the writer.
Work Queue Info	Use to view information on a writer, what items it picked up and the workers. From this page, you also have the ability to notify, start and stop workers across the cluster.

## Workflow Debugging, Logging, and Testing

For more information on application logging, see “Configuring Logging” on page 21 in the *System Administration Guide*.

Debugging a workflow is a more challenging task than debugging the standard BillingCenter 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 BillingCenter. (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 BillingCenter only commits these log message with the bundle.

Use the following logging method, for example, in an **Enter Script** block to log the current workflow step:

```
Workflow.log(summary, description)
```

The method returns the log entry (**WorkflowLogEntry**) that you can use for additional processing:

```
var workflowLog = Workflow.log("short description", "stack trace ...")
var summary = workflowLog.summary
```

### Process Logging

The following logging categories can be useful:

Category	Use for
WorkQueue	A category for general logging from the work queue.
WorkQueue.Instrumented	Capturing of runner state for a specific execution of the runner.
WorkQueue.Item	Logging (by workers) of each work item executed at the “info” level.
WorkQueue.Runner	Logging runners.

To write every message logged by every workflow, set the logging level of the workflow logger category to DEBUG (using **logging.properties**). The directive in the **logging.properties** file is:

```
log4j.category.Server.workflow=DEBUG
```

### Workflow Testing

It can often be difficult to test a workflow. This is especially true for one that is asynchronous and that requires the workflow to wait a specific amount of time before advancing to the next step. To facilitate testing, Guidewire BillingCenter supports a testing clock that permits the advancing of time (for development-mode servers only). If you have permission, you can access this functionality from the (unsupported) BillingCenter **Internal Tools** page. Depending on which clock you define in the **ITestingClock.xml** plugin registration file, you can do one of the following:

- Increment the current clock by a given period.
- Change the setting of the clock to a specific time. The clock remains at that time until another specific time is set.

### To enable the **ITestingClock** plugin

If the **ITestingClock** plugin is not already implemented, then you need to implement it.

1. In Studio, navigate to **Plugins** → **gw** → **plugin** → **system**.
2. Right-click **ITestingClock** and select **Implement**.
3. Click **Add** → **Java**.
4. Enter the following in the **Class** field.

```
com.guidewire.pl.plugin.system.internal.OffsetTestingClock
```



# Defining Activity Patterns

This topic discusses activity patterns, what they are, and how to configure them.

This topic includes:

- “What is an Activity Pattern?” on page 373
- “Pattern Types and Categories” on page 374
- “Using Activity Patterns in Gosu” on page 375
- “Calculating Activity Due Dates” on page 376
- “Configuring Activity Patterns” on page 376
- “Using Activity Patterns with Documents and Emails” on page 378
- “Localizing Activity Patterns” on page 378

## What is an Activity Pattern?

Activity patterns standardize the way that Guidewire BillingCenter creates activities. Activity patterns describe the kinds of activities that people perform while handling billing activity within an organization. For example, generating a letter of credit 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 an account, BillingCenter 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 accounts business processes from the **Administration** tab in BillingCenter. It is possible to create activities from activity patterns in different ways:

- You can manually create activities in BillingCenter.
- A business rule or some other Gosu code create activities as part of generating workplans or while responding to escalations, account exceptions, or other events.

- BillingCenter 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 the **New Assigned Activity** menu from the **Actions** menu.

---

**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 BillingCenter assigns an activity. Instead, activity assignment methods in the assignment rules or in Gosu expressions control how BillingCenter assigns an activity. Using the pattern name, the assignment methods determine to whom to assign the activity.

## Pattern Types and Categories

BillingCenter 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 BillingCenter 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 BillingCenter 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 other *internal* activity pattern types in the base configuration. All pattern types other than *General* are internal. Only internal BillingCenter 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 248.

In the base configuration, Guidewire BillingCenter 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 BillingCenter interface. You use them only within Gosu and BillingCenter 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 BillingCenter **New Activity** menu. The activity pattern categories appear only within the BillingCenter 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 BillingCenter interface.

BillingCenter displays the activity categories in the [New Activity Pattern](#) editor screen.

## Using Activity Patterns in Gosu

An activity pattern is a group of properties used to initialize a newly-created activity. The activity pattern acts as a starting template from which activity properties are initialized.

**Note:** Activity patterns can also be used to initialize shared activities (that is, activities having no single owner). This section improves readability by referring only to activities. However, all operations and behaviors of activity patterns apply equally to shared activities.

When an activity is created, it can optionally be associated with an activity pattern. If associated with a pattern, the property settings of the activity are initialized to the settings defined in the pattern.

The use of an activity pattern is optional. Alternatively, the properties of an activity can be set individually, without the use of an activity pattern. An activity pattern is simply a convenient method of initializing activity properties to common default values.

Each activity can be associated with one activity pattern. Multiple activity objects may be initialized using the same activity pattern.

Activity patterns are created in the BillingCenter application. Each activity pattern is identified by a unique text string stored in the object's `Code` property.

Gosu code can retrieve an existing activity pattern by referencing the pattern's unique `Code` property. The retrieved pattern can then be associated with an activity to initialize the activity's properties, as demonstrated in the following code segment.

```
/* Functional method to retrieve an activity pattern */
var act = new Activity()
act.ActivityPattern = gw.api.web.admin.ActivityPatternsUtil.getActivityPattern("approval")
```

An equivalent, but best practice, method to retrieve the activity pattern is to isolate the call to the `getActivityPattern` method in a Gosu enhancement, as shown below.

```
/* Best practice method to retrieve an activity pattern */
/* Define an enhancement to retrieve the pattern */
enhancement getActivityPatternByCode( code : String ) : ActivityPattern {
    var activityPattern = gw.api.web.admin.ActivityPatternsUtil.getActivityPattern( code )
    return activityPattern
}
/* Call the enhancement to get the pattern */
var act = new Activity()
act.ActivityPattern = getActivityPatternByCode("approval")
```

After an activity has been initialized by a pattern, any of the activity properties set by the pattern can be reset to other values, as shown below.

```
/* Initialize an activity with a pattern (Uses the enhancement defined above) */
/* Note: Assume the pattern sets the Priority property to Priority.TC_URGENT */
act.ActivityPattern = getActivityPatternByCode("approval")
/* Reset the Priority property to the desired value for this activity */
act.Priority = Priority.TC_NORMAL
```

As mentioned earlier, an activity pattern can also be used to initialize a shared activity, as shown below.

```
/* Create a shared activity */
var sharedAct = new SharedActivity()
/* Initialize it with an activity pattern, using the enhancement defined above */
sharedAct.ActivityPattern = getActivityPatternByCode("approval")
/* Properties initialized by the pattern can be reset, just as with an Activity object */
sharedAct.Priority = Priority.TC_LOW
```

## 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 BillingCenter calculates the due date for an activity. BillingCenter can calculate a target due date in hours or days. BillingCenter calculates due dates using the following pieces of information:

- **How much time?** How much time to take or how many hours or days to allow to complete the activity. You specify this using the **Target days** or **Target hours** value.
- **What is the starting point?** What point in time does BillingCenter use as the start point in calculating the target date? You specify this using the **Target start point** field.
- **What days to count?** BillingCenter can count calendar days or only business days. You specify this with the **Include these days** field.

BillingCenter 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. BillingCenter 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**, BillingCenter uses 0 for both. A target date is optional for activities.

### Escalation Dates

While the target date can indicate a service-level target (for example, complete within five business days), there can possibly be some later deadline after which the work becomes dangerously late. (This can be, for example, a 30 day state deadline.) BillingCenter 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, BillingCenter 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.

BillingCenter 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, BillingCenter uses 0 (zero) for both. An escalation date, like a target date, is optional for activities.

## Configuring Activity Patterns

BillingCenter 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 BillingCenter **Administration** tab. You can access the `activity-patterns.csv` file through Guidewire Studio by navigating to the `configuration → config → import → gen` folder.

---

**IMPORTANT** Do not remove any internal (non-General type) activity patterns or change their type, category, or code values. Internal BillingCenter application code requires them. You can change other fields associated with these types, however.

---

The **ActivityPattern** object contains the following properties:

Property	User interface field	Description
ActivityClass	Not Applicable	Indicates whether the activity is a task or an event. A task has a due date. An event does not.
AutomatedOnly	Not Applicable	Boolean value to specify whether the activity pattern is used only by automated additions (by business rules) to the workplan. Default value is false. If true, the activity pattern does not appear as a choice in the user interface.  For all activity patterns with a non-general type, Guidewire recommends setting this flag to true to ensure they are not visible in the user interface.
Category	Category	The category for grouping <b>ActivityPatterns</b> in the BillingCenter interface.
Code	Code	Any unique text <i>with no spaces</i> . Maximum length is 60 characters. This property is required. The Code property is used to identify the activity pattern when accessing the pattern in rules or Gosu code. You can see this value only through the <b>Administration</b> tab.
Command	Not Applicable	<i>Do not use.</i> For Guidewire use only.
Description	Description	Describes the expected outcome at the completion of this activity. It is visible only if you view the details of the activity.
DocumentTemplate	Document Template	Document template to display if you choose this activity. Enter the document template ID.
EmailTemplate	Email Template	Email template to display if you choose this activity. Enter the email template file name.
EscalationDays	Escalation days	The number of days from the <b>escalationstartpt</b> to set the <b>Escalation Date</b> for an activity.
EscalationHours	Escalation hours	The number of hours from the <b>escalationstartpt</b> to set the <b>Escalation Date</b> for an activity.
EscalationInclDays	Include these days	Specifies which days to include. You can set this <b>businessdays</b> or <b>elapsed</b> .
EscalationStartPt	Escalation start point	The initial date used to calculate the target date. If you specify <b>escalationdays</b> or <b>escalationhours</b> , you need to specify this parameter. Otherwise, this parameter is optional.
Mandatory	Mandatory	Boolean value indicating whether completion of the activity is mandatory. This property is required. Default value is false. Non-mandatory activities are optional tasks.
Priority	Priority	Used to sort more important activities to the top of a list of work. This property is required. You can set this property to the following values: <ul style="list-style-type: none"><li>• <b>urgent</b></li><li>• <b>high</b></li><li>• <b>normal</b></li><li>• <b>low</b></li></ul>
ShortSubject	Short Subject	A brief description of the activity used on small areas of the BillingCenter interface such as a calendar event entry. Maximum length of 10 characters.
Subject	Subject	A short text description of the activity that BillingCenter shows in activity lists. This property is required.
TargetDays	Target days	The number of days from the <b>targetstartpoint</b> to set the activity's <b>Target Date</b> .
TargetHours	Target hours	The number of hours from the <b>targetstartpoint</b> to set the activity's <b>Target Date</b> .
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"><li>• <b>elapsed</b>—count all Calendar days</li><li>• <b>businessdays</b>—count all Business days as defined by the business calendar</li></ul>

Property	User interface field	Description
TargetStartPoint	Target start point	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.
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 BillingCenter 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 BillingCenter 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, BillingCenter does the following:

- If you create a new activity from this activity pattern, BillingCenter automatically populates any template field for which you specified a template with the name of that template.
- If you then open this activity, BillingCenter 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, BillingCenter 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 376 for details of working with the `activity-patterns.csv` file.

## Localizing Activity Patterns

BillingCenter 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 BillingCenter interface—if you configure BillingCenter for multiple locales:

- Subject

If you configure BillingCenter 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 text for that activity pattern.

**See also**

- For information on how to make a database column localizable (and thus, an object property localizable), see “Localizing Administration Data” on page 67 in the *Globalization Guide*.



# Testing Gosu Code



# Testing and Debugging Your Configuration

After you use Guidewire Studio to make configuration changes to your application, you will typically want to run the application to test those changes. Guidewire Studio provides powerful features to help you make sure that your application works the way you intend.

This topic includes:

- “Testing BillingCenter With Guidewire Studio” on page 383
- “The Studio Debugger” on page 386
- “Setting Breakpoints” on page 386
- “Stepping Through Code” on page 387
- “Viewing Current Values” on page 388
- “Resuming Execution” on page 389
- “Using the Gosu Scratchpad” on page 389
- “Suggestions for Testing Rules” on page 390

## Testing BillingCenter With Guidewire Studio

After you make configuration changes to BillingCenter, you can start it directly from within Guidewire Studio and test your changes. You can also use the powerful debugging features that Studio provides.

This topic contains:

- “Running BillingCenter Without Debugging” on page 384
- “Debugging BillingCenter Within Studio” on page 384
- “Debugging a BillingCenter Server That Is Running Outside of Studio” on page 384

## Running BillingCenter Without Debugging

If you do not plan to use debugging features, then you can run BillingCenter directly from within Studio to quickly test your configuration changes. Running BillingCenter this way is similar to starting it from the command line with the command `gwbc dev-start`, but without having to switch out of Studio.

### To run BillingCenter without debugging

- In Studio, click Run → Run 'Server'.

The BillingCenter server starts, and debug messages appear in the Run tool window in Studio.

## Debugging BillingCenter Within Studio

You can run BillingCenter in the Studio debugger, which provides additional features to help you verify that your configuration changes are working as desired.

### To debug BillingCenter within Studio

- In Studio, click Run → Debug 'Server'.

The BillingCenter server starts, and debug messages appear in the Debug tool window in Studio.

#### See also

- “The Studio Debugger” on page 386

## Debugging a BillingCenter Server That Is Running Outside of Studio

Instead of running BillingCenter within the Studio debugger, you can have the debugger connect to a BillingCenter server that is running outside of Studio. The debugger can connect to a BillingCenter server running either on the local computer or on a remote computer.

You must choose one of the following ways for Studio to connect to the server:

Connection type	Description	See
shared memory	Connects Studio to a server running on the same computer; faster than a socket connection.	“Debugging a BillingCenter Server Using a Shared Memory Connection” on page 384
socket	Allows Studio to connect to a server running on a remote computer; slower than a shared memory connection.	“Debugging a BillingCenter Server Using a Socket Connection” on page 385

### Debugging a BillingCenter Server Using a Shared Memory Connection

Studio can use a shared memory connection to connect to a BillingCenter server running on the same computer. You must manually start the server in the proper debug mode, and create the proper debug configuration in Studio.

1. Start the BillingCenter server.
  - a. Start the server using the following command:  
`gwbc dev-debug-shmem`
  - b. Towards the beginning of the server console message output, look for the label `dev-debug-shmem:`, and then the text that is similar to the following.  
`Listening for transport dt_shmem at address: javaAddress`
  - c. Make a note of the string provided for `javaAddress`.

2. Create the debug configuration in Studio.
  - a. In Studio, click Run → Edit Configurations.
  - b. In the Run/Debug Configurations dialog, click Add New Configuration , and then click Remote.
  - c. In the Name text box, type a name for the configuration. For example, server-shmem.
  - d. For the Transport option, click Shared memory.
  - e. In the Shared Memory Address text box, type the *javaAddress* that you noted when you started the server.
  - f. Click OK.
3. Connect the Studio debugger to the BillingCenter server using the shared memory connection.
  - a. In Studio, in the Select Run/Debug Configuration drop-down list, select the debug configuration that you created.
  - b. Click Run → Debug '*configName*', where *configName* is the name of the debug configuration that you created.

The debugger connects to the BillingCenter server, and debug messages appear in the Debug tool window in Studio.

### Debugging a BillingCenter Server Using a Socket Connection

Studio can use a socket connection to connect to a BillingCenter server running either on the same computer or a remote computer. If running on the same computer, however, a shared memory connection is faster than a socket connection.

You must manually start the server in the proper debug mode, and create the proper debug configuration in Studio.

1. Start the BillingCenter server.
  - a. Start the server using the following command:

```
gwbc dev-debug-socket
```
  - b. Towards the beginning of the server console message output, look for the label dev-debug-socket:, and then the text that is similar to the following.

```
Listening for transport dt_socket at address: portNumber
```
  - c. Make a note of the value provided for *portNumber*.
2. Create the debug configuration in Studio.
  - a. In Studio, click Run → Edit Configurations.
  - b. In the Run/Debug Configurations dialog, click Add New Configuration , and then click Remote.
  - c. In the Name text box, type a name for the configuration. For example, server-socket.
  - d. For the Transport option, click Socket.
  - e. In the Host text box, type the hostname of the computer on which the BillingCenter server is running.
  - f. In the Port text box, type the *portNumber* that you noted when you started the server.
  - g. Click OK.
3. Connect the Studio debugger to the BillingCenter server using the shared memory connection.
  - a. In Studio, in the Select Run/Debug Configuration drop-down list, select the debug configuration that you created.
  - b. Click Run → Debug '*configName*', where *configName* is the name of the debug configuration that you created.

The debugger connects to the BillingCenter server, and debug messages appear in the **Debug** tool window in Studio.

### Debugging a BillingCenter Server in Suspended Mode

When you start a BillingCenter server in debug mode, the server completes its full startup process until it becomes ready for client connections. In this case, you cannot begin to debug the server until it is ready. However, if the behavior that you want to debug occurs earlier in the startup process, then you can start the server in *suspended* mode instead. In suspended mode, the BillingCenter startup pauses as soon as its Java process is initially established. The startup process continues only once you start the Studio debugger.

To start the BillingCenter server in suspended mode, use one the following commands instead of the debug commands:

- `gwbc dev-suspend-shmem`
- `gwbc dev-suspend-socket`

## The Studio Debugger

Guidewire Studio includes a code *debugger* to help you verify that your Gosu code is working as desired. It works whether the code is in a Gosu rule, a Gosu class, or a BillingCenter PCF page. You access this functionality through the Studio **Run** menu and through specific debug icons on the Studio toolbar. You must be connected to a running BillingCenter server to use the Studio debugger. (If you do not have a connection to a running server, Studio attempts to run one.) If the debugger is active, you can debug Gosu code that runs in the Gosu Scratchpad and Gosu code that is part of the running application.

If instructed, Studio can pause (at a breakpoint that you set) before it runs a specified line of code. This can be any Gosu code, whether contained in a rule or a Gosu class. The debugger can also run on Gosu that you call from a PCF page, if the called code is a Studio class.

After Studio pauses, you can examine any variables or properties used by Gosu and view their values at that point in the debugger pane. You can then have Studio continue to step through your code, pausing before each line. This allows you to monitor values as they change, or simply to observe the execution path through your code.

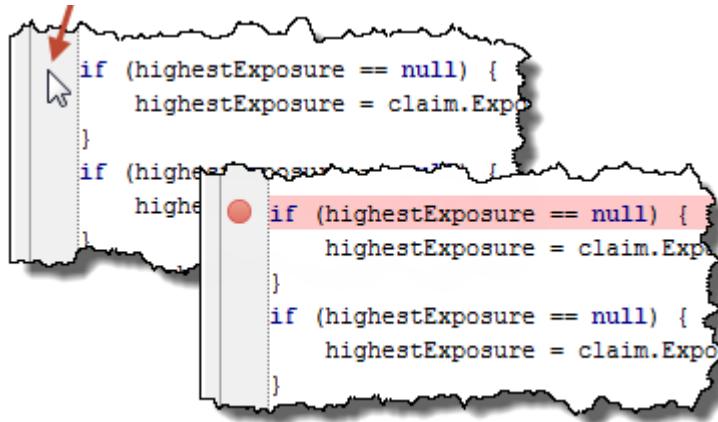
**Note:** Do not perform debugging operations on a live production server.

## Setting Breakpoints

A breakpoint is a place in your code at which the debugger pauses execution, giving you the opportunity to examine current values or begin stepping through each line. The debugger pauses before executing the line containing the breakpoint. The debugger identifies a breakpoint by highlighting the related line of code and placing a breakpoint symbol  next to it.

### To set a breakpoint

Place the cursor on the line of code on which to set the breakpoint, and then on the Run menu, click **Toggle Line Breakpoint**. You can also click in the gray column next to a code line:



You can set multiple breakpoints throughout your code, with multiple breakpoints in the same block of code, or if desired, breakpoints in multiple code blocks. The debugger pauses at the first breakpoint encountered during code execution. After it pauses, the debugger ignores other breakpoints until you continue normal execution.

You can set a breakpoint in a rule condition statement, as well. You cannot set a breakpoint on a comment.

### To view a breakpoint

On the Run menu, click **View Breakpoints**. Selecting this menu item opens that **View Breakpoints** dialog in which you can do the following:

- View all of your currently set breakpoints
- Deactivate any or all of your breakpoints (which makes them non-functional, but does not remove them from the code)
- Remove any or all breakpoints
- Navigate to the code that contains the breakpoint

Thus, from this dialog, you can deactivate, remove, or add a breakpoint.

### To remove a breakpoint

Place the cursor on the line of code containing the breakpoint to remove, and then on the Run menu, click **Toggle Line Breakpoint**. You can also click the breakpoint symbol next to the line of code.

## Stepping Through Code

After the debugger pauses execution, you can step through the code one line at a time in one of the following ways:

<b>Step over</b>	Execute the current line. If the current line is a method call, then run the method and return to the next line in the current code block after the method ends. To step through your code in this manner, on the Run menu, click <b>Step Over</b>  .
<b>Step into</b>	Execute the current line of code. If the current line is a method call, then step into the method and pause before executing the first line for that method. To step through your code in this manner, on the Run menu, click <b>Step Into</b>  .

The only difference between these two stepping options is if the current line of code is a method call. You can either *step over* the method and let it run without interruption, or you can *step into* it and pause before each line. For other lines of code that are not methods, stepping over and stepping into behave in the same way.

While paused, you can navigate to other rules or classes if you want to look at their code. To return to viewing the current execution point, on the Run menu, click **Show Execution Point** . Studio highlights the line of code to execute the at the next step.

## Viewing Current Values

After the debugger pauses at a breakpoint in your code, you can examine the current values of variables or entity properties. You can watch these values and see how they change as each line of code executes.

This topic includes:

- “Enabling the Viewing of Entities While Debugging” on page 388
- “Viewing Variables” on page 388
- “Defining a Watch List” on page 388

### Enabling the Viewing of Entities While Debugging

Collecting and displaying real-time entity information can make the server run slower, so you can turn this feature on and off as needed.

#### To enable the viewing of entities

1. In Guidewire Studio, click .
2. Navigate to the Guidewire Studio page, and then select **Enhance Entities Visualization**.
3. When the server pauses at a breakpoint, look in the **Debug** pane, and then in the **Variables** or **Watches** list. Locate the entity that you want to view.
4. Right-click the entity, and then click **View as → Entity**.

### Viewing Variables

The **Debugger** tab of the **Debug** pane shows the root entity that is currently available. For example, in activity assignment code, the root entity is an **Activity** object. To view any property of the root entity, expand it until you locate the property.

The **Debugger** tab also shows the variables that you have currently defined. Note, however, that you can view only the entities and variables that are available in the current *scope* of the code. Suppose, for example, that an **Activity** entity is available in an activity assignment class. However, if that class calls a different class and you step into that class, the **Activity** entity is no longer part of the scope. Therefore, it is no longer available. (Unless, you pass the **Activity** object in as a parameter.)

### Defining a Watch List

After executing each line of code, Studio resets the list of values shown in the **Debug** frame. In doing so, it collapses any property hierarchies that you may have expanded. It would be inconvenient for you to expand the hierarchy after each line and locate the desired properties all over again.

Instead, you can define a watch list containing the Gosu expressions in which you have an interest in monitoring. The debugger evaluates each expression on the list after each line of code executes, and shows the result for each expression on the list. For example, you can add `Activity.AssignedUser` to the watch list and then monitor the list as you step through each line of code. If the property changes, you see that change reflected immediately on the list. You can also add variables to the list so you can monitor their values, as well.

To add an expression to the watch list, in the **Variables** pane, right-click the expression, and then click **Add to Watches**.

As you step through each line of code in the debugger, keep the **Watches** pane visible so that you can monitor the values of the items on the list.

## Resuming Execution

After the debugger pauses execution of your code, and after you are through performing any necessary debugging steps, you may want to resume normal execution. You can do this in the following ways:

<b>Stop debugging</b>	Resume normal execution, and ignore all remaining breakpoints. To stop debugging, click <b>Mute Breakpoints</b>  , and then click <b>Resume Execution</b>  .
<b>Continue debugging</b>	Resume normal execution, but pause again at the next breakpoint, if any. To continue debugging, click <b>Resume Execution</b>  without having <b>Mute Breakpoints</b>  set.

## Using the Gosu Scratchpad

You use the Gosu Scratchpad to execute Gosu programs and evaluate Gosu expressions. Instead of needing to perform BillingCenter operations to trigger your Gosu code, you can run your code directly in the Scratchpad and see immediate results.

To run the Gosu Scratchpad, click **Tools** → **Gosu Scratchpad** .

To execute queries against the database in Studio or the Gosu Scratchpad, you must first connect to a running application server. The result of a query is always a read-only query object. To work with this read-only object, you must add it to a transaction bundle. See also “Using the Results of Find Expressions (Using Query Objects)” on page 193 in the *Gosu Reference Guide*.

You can test the following in the Gosu Scratchpad:

Code	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.

## Executing Code in the Gosu Scratchpad

### To execute code in the Gosu Scratchpad

- Type your code, and then click **Run** .

## Accessing Application Data in the Gosu Scratchpad

You can always use the Gosu Scratchpad to test generic Gosu code. If the Scratchpad code needs to access application data, then the BillingCenter server must be running. For example, the following code accesses group and user entities, and therefore requires access to the application server:

```
var group : Group = Group(2)
for (var member in group.Members) {
    print(member.User.Contact.LastName)
}
```

To access application data in the Gosu Scratchpad, run the BillingCenter server in debug mode, and then instruct the Scratchpad to run your code in that process. For information on running the BillingCenter server in debug mode, see “Testing BillingCenter With Guidewire Studio” on page 383.

### To run Scratchpad code in the application server debug process

- Type your code, and then click **Run in Debug Process** .

## Suggestions for Testing Rules

Guidewire recommends that you practice the following simple suggestions to make testing and debugging your rules a straightforward process:

- Enter one rule at a time and monitor for syntax correctness—check the green light (at the bottom of the pane) before starting a new rule.
- Enter rules in the order in which you want the debugger to evaluate them: **Condition** and then **Action**.
- Maintain two sessions while testing. As you complete and save each rule in Studio, toggle to an open BillingCenter 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 BillingCenter logging settings.

# Using GUnit

You use Studio GUnit to configure and run repeatable tests of your Gosu code in a similar fashion as JUnit works with Java code. (GUnit is similar to JUnit 3.0 and compatible with it.) GUnit works automatically and seamlessly with the embedded QuickStart servlet container, enabling you to see the results of your GUnit Gosu tests within Studio.

GUnit provides a complete test harness with base classes and utility methods. You can use GUnit to test any body of Gosu code except for Gosu written as part of Rules. (To test Gosu in Rules, use the Studio debugger. See “Testing and Debugging Your Configuration” on page 383 for details.)

This topic includes:

- “The TestBase Class” on page 391
- “Configuring the Server Environment” on page 393
- “Configuring the Test Environment” on page 394
- “Creating a GUnit Test Class” on page 396
- “Using Entity Builders to Create Test Data” on page 398

**Note:** Guidewire does not recommend or support the use of classes that extend `gw.api.databuilder.DataBuilder` or classes that reside in the `gw.api.databuilder.*` package in a production environment. Guidewire provides GUnit as a development test facility only.

## The TestBase Class

Guidewire uses the `TestBase` class as the root class for all GUnit tests. Your test class must extend the Guidewire `TestBase` class. This class provides the following:

- The base test infrastructure, setting up the environment in which the test runs.
- A set of `assert` methods that you can use to verify the expected result of a test.
- A set of `beforeXX` and `afterXX` methods that you can override to provide additional testing functionality (for example, to set up required data before running a test method).

The `TestBase` class interacts with an embedded QuickStart servlet container in running your GUnit tests. This class has access to all of the embedded QuickStart server files and servlets. (GUnit starts and stops the embedded QuickStart servlet container automatically. You have no control over it.) This class also initializes all server dependencies.

## Overriding TestBase Methods

Guidewire exposes two groups of `beforeXX` and `afterXX` methods in the `TestBase` class that you can use to perform certain actions before and after the tests execute. These methods are a way to set up any required dependencies for tests and to clean up after a test finishes.

To use one of these methods, you need to provide an overridden implementation of the method in your test class.

- Use `beforeClass` to perform some action before GUnit instantiates the test class.
- Use `afterClass` to perform some action after all the tests complete but before GUnit destroys the class.
- Use `beforeMethod` to perform some action before GUnit invokes a particular test method.
- Use `afterMethod` to perform some action after a test method returns.

These methods have the following signatures.

```
beforeClass() throws Exception {...}
afterClass() {...}
beforeMethod() throws Exception {...}
afterMethod(Throwable possibleException) {...} //If the test resulted in an exception, parameter
//possibleException contains the exception.
```

### Initializing static and instance variables in a TestBase test class

When implementing a `TestBase` test class and running that class in Studio, do not use static initializers or initialize variables in a constructor. Instead, initialize variables on the class as follows:

- Initialize static variables on the class by overriding the `beforeClass` method and initializing the variable in that method.
- Initialize instance variables by overriding the `beforeMethod` method and initializing the variable in that method.

For example, when initializing a class that uses the `typekey` type:

```
class MyTypekeyRelatedTest extends TestBase
    static var _transCurrency : Currency

    override function beforeClass() {
        super.beforeClass() // must call super method!
        _transCurrency = Currency.TC_EUR
    }
}
```

The following is another example for initializing a class that uses a third-party library:

```
class MyComplicatedTest extends TestBase
    var _testHelper : ThirdPartyClass

    override function beforeMethod() {
        super.beforeClass() // must call super method!
        _testHelper = new ThirdPartyClass()
    }
}
```

### 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 398 for details on how to create test data.
- See “Creating a Builder for a Custom Entity and Testing It” on page 407 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.

BillingCenter interprets any class that contains the annotation `@ServerTest` immediately before the class definition as a server test. If you create a test class through Guidewire Studio, then Studio automatically adds the server runtime annotation `@ServerTest` immediately before the class definition. At the same time, Studio also adds `extends gw.testharness.TestBase` to the class definition. All GUnit tests that you create must extend this class. (See the “The TestBase Class” on page 391 for more information on this class.)

Although Studio automatically adds the `@ServerTest` annotation to the class definition, it is possible to remove this annotation safely. As the `TestBase` class already includes this annotation, Guidewire does not explicitly require this annotation in any class that extends the `TestBase` class.

By default, the server starts at a run level set to `Runlevel.NO_DAEMONS`. To change this default, see the description of the `@RunLevel` annotation in the next section.

### Server Environment

Environment tags provide additional functionality. You use environment tags to replace functionality specific to an external environment. This can include defining new SOAP endpoints or creating tests for custom PCF page, for example.

Guidewire provides the following environment tags for use in GUnit tests.

Annotation (@gw.testharness.*)	Description
<code>@ChangesCurrentTime</code>	Sets up a mock system clock that allows the test to change the current time during the test.
<code>@ProductUnderTest</code>	Explicitly sets the product being tested. Typically, Studio infers this from the test class package. However, you can use this annotation if that is not possible, as with <code>gw.api</code> tests, for example.

Annotation (@gw.testharness.*)	Description
@ProductionMode	<p>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).</p>
	<p>You can check the server mode in Gosu, using the following:</p> <pre data-bbox="714 451 1204 481">gw.api.system.server.ServerModeUtil.isDev()</pre>
@RealPCFs	<p>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.</p>
@RunInDatabase	<p>Defines the databases against which to run this class’s tests. Without this annotation, this class only runs in H2 suites. The annotation takes an array of DatabaseForTest values, specifying the databases which are specifically to be tested, or DatabaseForTest.ALL that allows the class to be run against any database.</p>
@RunLevel	<p>Allows a test to run at a different run level. The default value is RunLevel.NONE. You can, however, change the run level to one of the following (although each level takes a bit more time to set up):</p> <ul style="list-style-type: none"> <li>• Runlevel.NONE - Use if you do not want any dependencies at all.</li> <li>• Runlevel.SHUTDOWN - Use if you want all the basic dependencies set up, but with no database connection support.</li> <li>• Runlevel.NO_DAEMONS - Use for a normal server startup without background tasks. (This is also suitable for SOAP tests.)</li> <li>• Runlevel.MULTIUSER - Use to start a complete server (batch process, events, rules, Web requests, and all similar components).</li> </ul>

## Configuring the Test Environment

You define the run and debug parameter settings for a GUnit test class through the **Run/Debug Settings** dialog, which you can access in any of the following ways:

- Click **Run** → **Edit Configurations**.
- On the main toolbar, in the **Select Run/Debug Configuration** drop-down list, click **Edit Configurations**.
- In the **Project** tool window, right-click the test package, and then click **Create 'Tests in 'PackageName'**.
- Open the test class in the editor, right-click anywhere in the method, and then click **Create 'testName()'**.

You can set various default configuration parameters for all tests, or configure parameters for a particular test.

### Setting Default Configuration Parameters for All Tests

It is possible to set a number of default configuration parameters that GUnit uses for all tests. To do this, in the **Run/Debug Configurations** dialog, expand **Defaults**, and then click **Junit**. Enter the default configuration parameters as appropriate. See “Configuration Parameters” on page 395 for a description of the various configuration parameters.

### Adding a Named Set of Configuration Parameters

It is possible to create a defined set of configuration parameters to use with one or more tests. To do this, first add that configuration under the **Application** section of the list in the **Run/Debug Configurations** dialog. Use the following dialog toolbar icons.

Icon	Use to
	Add a new named test configuration to the list. Click this, and then click JUnit.
	Delete the selected configuration from the list.
	Clone the selected test configuration.
	Move the selected configuration up within the list.
	Move the selected configuration down within the list.

### Viewing Configuration Settings Before Launching

It is possible to turn on, or off, the **Run/Debug Configurations** dialog before running a test. To view the GUnit configuration settings before launching a test, expand the **Before launch** section of that dialog, and then set the **Show this page** check box.

If you unset this option, you do not see the **Run/Debug Configurations** dialog upon starting a test. Instead, the test starts immediately. In addition, selecting **Run** or **Debug** from the Studio **Run** menu does not open this dialog either. To access the **Run/Debug Configurations** dialog again, click **Run → Edit Configurations**.

## Configuration Parameters

Use the **Run/Debug Configurations** dialog to enter the following configuration parameters:

- Name
- Test Kind
- VM Options

You may not see some of the parameter fields until you actually load a test into Studio and select it. See “Creating a GUnit Test Class” on page 396 for information on how to create a GUnit test within Guidewire Studio.

### Name

If desired, you can set up multiple run and debug GUnit configurations. Each named configuration represents a different set of run and debug startup properties. To create a new named configuration:

- Click **Add** and create a new blank configuration.
- Select an existing configuration, then click **Copy Configuration** to copy the existing configuration parameters to the new configuration.
- Select the test class in the Project window, and then click either **Run 'TestName'** or **Debug 'TestName'**. Then, select the name of the test from the list of GUnit tests and click **OK**. This has an advantage of populating the fully qualified class name field.

After you add the new configuration node on the left-hand side, you can enter a name for it on the right-hand side of the dialog.

### Test Kind

Use to set whether to test all the classes in a package, a specific class, or a specific method in a class. The text entry field changes as you make your selection.

- For **All in Package**, enter the fully qualified package name. Select this option to run all GUnit tests in the named package.
- For **Class**, enter the fully qualified class name. Select this option to run all GUnit tests in the named class.
- For **Method**, enter both the fully qualified class name and the specific method to test in that class.

### VM Options

Use to set parameters associated with the JVM and the Java debugger. To set specific parameters for the JVM to use while running this configuration, enter them as a space separated list in the **VM Options** text box. For example:

```
-client -Xmx700m -Xms200m -XX:MaxPermSize=100m -ea
```

You can change the JVM parameters based on the test. For example, while testing a large class or while running numerous test methods within a class, you may want to increase your maximum heap size.

## Creating a GUnit Test Class

The following is an example of a GUnit test class. Use this sample code as a template in creating your own test classes.

```
package AllMyTests

uses gw.testharness.TestBase
@gw.testharness.ServerTest
class MyTest extends TestBase {

    construct(testname : String) {
        super(testname)
    }
    ...
    function testSomething() {
        //perform some test
        assertEquals("reason for failure", someValue, someOtherValue)
    }
    ...
}
```

Notice the following:

- The test class exists in the package `AllMyTests`. Thus, the full class path is `Tests.AllMyTests.MyTest`. You must place your test classes in the `modules/configuration/gtest` folder. You are free, however, to name your test subpackages as you choose.
- The class file name and the class name are identical and end in `Test`.
- The test class extends `TestBase`.
- The class definition files contains a `@ServerTest` annotation immediately before the class definition.
- The class definition contains a `construct` code block. This code block can be empty or it may contain initialization code.
- The class definition contains one or more test methods that begin with the word `test`. The word `test` is case-sensitive. For example, GUnit will recognize the string `testMe` as a method name, but not the string `TestMe`.
- The test method contains one or more `assert` methods, each of which “asserts” an expected result on the object under test.

## Server Tests

You specify the type of test using annotations. Currently, Guidewire supports server tests only. Server tests provide all of the functionality of a running server. You must include the @ServerTest annotation immediately before the test class definition to specify that the test is a server test. See “Configuring the Server Environment” on page 393 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, JUnit generates an error.) Each test method uses a verification method to test a single condition. For example, a method can test if the result of some operation is equal to a specific value. In the base configuration, Guidewire provides a number of these verification methods. For example:

- `assertTrue`
- `assertEquals`
- `verifyTextInPage`
- `verifyExists`
- `verifyNull`
- `verifyNotNull`

Many of these methods appear in multiple forms. Although there are too many to list in their entirety, the following are some of the basic `assert` methods. To see a complete list of these methods in their many forms, use the code completion feature in Studio.

```
assertArrayDoesNotContain  
assertArrayEquals  
assertBigDecimalEquals  
assertBigDecimalNotEquals  
assertCollection  
assertCollectionContains  
assertCollectionDoesNotContain  
assertCollectionContains  
assertCollectionSame  
assertComparesEqual  
assertDateEquals  
assertEmpty  
assertEquals  
assertEqualsIgnoreCase  
assertEqualsIgnoreLineEnding  
assertEqualsUnordered  
assertFalse  
assertFalseFor  
assertGreaterThan  
assertIteratorEquals  
assertIteratorSame  
assertLength  
assertList  
assertListEquals  
assertListSame  
assertMethodDeclaredAndOverridesBaseClass  
assertNotNull  
assertNotSame  
assertNotZero  
assertNull  
assertSame
```

```
assertSet
assertSize
assertSuiteTornDown
assertThat
assertTrue
assertTrueWithin
assertZero
```

### The assertThat Method

Choosing the `assertThat` method opens up a whole variety of different types of assertions, dealing with strings, collections, and many other object types. To see a complete list of this method in its many forms, use the code completion feature in Studio.

### Failure Reasons for Asserts

Guidewire strongly recommends that, as appropriate, you use an assert method that takes a string as its first parameter. For example, even though Guidewire supports both versions of the following assert method, the second version is preferable as it includes a failure reason.

```
assertEquals(a, b)
assertEquals("reason for failure", a, b)
```

Guidewire recommends that you document a failure reason as part of the method rather than adding the reason in a comment. The GUnit test console displays this text string if the assert fails, which makes it easier to understand the reason of a failure.

### To create a GUnit test class

1. In the Project window, navigate to **configuration** → **gtest**.
1. Right-click **gtest**, and then click **New** → **Package**.
2. In the **Enter new package name** text box, type the name of the package.
3. Right-click the new package, and then click **New** → **Gosu Class**.
4. In the **Name** text box, type the name of the test class. This class file name must match the test class name and both must end in “Test”. This action creates a class file containing a “stub” class. For example, if your class file is **MyTest.gs**, Studio populates the file with the following Gosu:

```
package demo

@gw.testharness.ServerTest
class MyTest extends gw.testharness.TestBase {
    construct() {
        ...
    }
    ...
}
```

### To run a GUnit test

1. In the Project window, navigate to **configuration** → **gtest**, and then to your test class.
2. Right-click the test, and then click either **Run 'TestName'** or **Debug 'TestName'**. This action opens a test console at the bottom of the screen.
3. (Optional) If desired, you can also create individual run/debug settings to use while running this test class. For details, see “Configuring the Test Environment” on page 394.

## Using Entity Builders to Create Test Data

**Note:** Guidewire does not recommend or support the use of classes that extend `gw.api.databuilder.DataBuilder` or classes that reside in the `gw.api.databuilder.*` package in a production environment. Guidewire provides GUnit as a development test facility only.

As you run tests against code, you need to run these test in the context of a known set of data objects. This set of objects is generally known as a *test fixture*. You use Gosu entity builders to create the set of data objects to use in testing.

Guidewire provides a number of entity “builders” as utility classes to quickly and concisely create objects (entities) to use as test data. The BillingCenter base configuration provides builders for the base entities (like AccountBuilder, for example). However, if desired, you can extend the base DataBuilder class to create new or extended entities. You can commit any test data that you create using builders to the test database using the `bundle.commit` method.

For example, the following builder creates a new Person object with a `FirstName` property set to “Sean” and a `LastName` property set to “Daniels”. It also adds the new object to the default test bundle.

```
var myPerson = new PersonBuilder()  
    .withFirstName("Sean")  
    .withLastName("Daniels")  
    .create()
```

For readability, Guidewire recommends that you place each configuration method call on an indented separate line starting with the dot. This makes code completion easier. It also makes it simpler to alter a line or paste a new line into the middle of the chain or to comment out a line.

Gosu builders extend from the base class `gw.api.databuilder.DataBuilder`. To view a list of valid builder types in Guidewire BillingCenter, use the Studio code completion feature. Enter `gw.api.databuilder.` in the Gosu editor and Studio displays the list of available builders.

### Package Completion

As you create an entity builder, you must either use the full package path, or add a `uses` statement at the beginning of the test file. However, in general, Guidewire recommends that you place the package path in a `uses` statement at the beginning of the file.

```
uses gw.api.builder.AccountBuilder  
  
@gw.testharness.ServerTest  
class MyTest extends TestBase {  
  
    construct(testname : String) {  
        super(testname)  
    }  
    ...  
    function testSomething() {  
        //perform some test  
        var account = new AccountBuilder().create()  
    }  
    ...  
}
```

Or, more simply (although Guidewire does not recommend this), enter the full path within the test class itself:

```
var account = new gw.api.builder.AccountBuilder().create()
```

## Creating an Entity Builder

To create a new entity builder of a particular type, you merely need to use the following syntax:

```
new TypeOfBuilder()
```

This creates a new builder of the specified type, with the Builder class setting various default properties on the builder entity. (Each entity builder provides different default property values depending on its particular implementation.) For example, to create (or build) a default address, use the following:

```
var address = new AddressBuilder()
```

To set specific properties to specific values, you need to also use the property configuration methods. There are three different types of property configuration methods, each which serves a different purpose as indicated by the method's initial word.

Initial word	Indicates
on	A link to a parent, for example, PolicyPeriod is on an Account, so the method is <code>onAccount(Account account)</code> .
as	A property that holds only a single state, for example, <code>asBusinessType</code> or <code>asAgencyBill</code> .
with	The single element or property to be set. For example, the following sets a <code>FirstName</code> property: <code>withFirstName("Joe")</code>

Use a `DataBuilder.with(...)` configuration method to add a single property or value to a builder object. For example, the following Gosu code creates a new `Address` object and uses a number of `with(...)` methods to initialize properties on the new object. It then uses an `asType(...)` method to set the address type.

```
var address = new AddressBuilder()
    .withAddressLine1( codeStr1 + " Main St." )
    .withAddressLine2( "Suite " + codeStr2 )
    .withCity( "San Mateo" )
    .withState( "CA" )
    .withPostalCode( "94404-" + codeStr3 )
    .asBusinessType()
    ...
```

After you create a builder entity, you are responsible for writing that entity to the database as part of a transaction bundle. In most cases, you must use one of the builder `create` methods to add the entity to a bundle. Which `create` method one you choose depends on your purpose.

To complete the previous example, you need to add a `create` method at the end.

```
var address = new AddressBuilder()
    .withAddressLine1( codeStr + " Main St." )
    ...
    .create()
```

## Builder Create Methods

The `DataBuilder` class provides the following `create` methods:

```
builderObject.create( bundle )
builderObject.create()
builderObject.createAndCommit()
```

The following list describes these `create` methods.

Method	Description
<code>create()</code>	Creates an instance of this builder's entity type, in the default bundle. This method does not commit the bundle. Studio resets the default bundle before every test class and method.
<code>createAndCommit()</code>	Creates an instance of this builder's entity type, in the default bundle and performs a commit of that default bundle.
<code>create(bundle)</code>	Creates an instance of this builder's entity type, with values determined by prior calls to the entity. The bundle parameter sets the bundle to use while creating this builder instance.

### The No-Argument Create Method

The no-argument `create` method uses a default bundle that all the builders share. This is adequate for most test purposes. However, as all objects created this way share the same bundle, committing the bundle on just one of the created objects commits all of the objects to the database. This also makes them available to the BillingCenter interface portion of a test. For example:

```
var address = new AddressBuilder()
    .withCity( "Springfield" )
    .asHomeAddress()
```

```
.create()  
new PersonBuilder()  
    .withFirstName("Sean")  
    .withLastName("Daniels")  
    .withPrimaryAddress(address)  
    .create()  
address.Bundle.commit()
```

In this example, `Address` and `Person` share a bundle, so committing `address.Bundle` also stores `Person` in the database. If you do not need a reference to the `Person`, then you do not need to store it into a variable.

JUnit resets the default bundle before every test class and method.

### The Create and Commit Method

The `createAndCommit` method is similar to the `create` method in that it adds the entity to the default bundle. It then, however, commits that bundle to the database.

### The Create with Bundle Method

If you need to work with a specific bundle, use the `create(bundle)` method. Guidewire recommends that you use this method inside of a transaction block. A transaction block provides the following:

- It creates the bundle at the same time as it creates the new builder.
- It automatically commits the bundle as it exits.

The following example illustrates the use of a data builder inside a transaction block.

```
function myTest() {  
    var person : Person  
  
    Transaction.RunWithNewBundle( \ bundle -> {  
        person = new PersonBuilder()  
            .withFirstName( "John" )  
            .withLastName( "Doe" )  
            .withPrimaryAddress( new AddressBuilder()  
                .withCity( "Springfield" )  
                .asHomeAddress() )  
            .create( bundle )  
    } )  
  
    assertEquals( "Doe", person.LastName )  
}  
}
```

Notice the following about this example:

- The example declares the `person` variable outside the transaction block, making it accessible elsewhere in the method.
- The data builder uses an `AddressBuilder` object nested inside `PersonBuilder` to build the address.
- The `Transaction.RunWithNewBundle` statement creates the bundle and automatically commits it after Gosu Runtime executes the supplied code block.

In summary, the `create(bundle)` method does not create a bundle. Rather, it uses the bundle passed into it. Guidewire recommends that you use this method inside a transaction block that both creates the bundle and commits it automatically.

If you do not use this method inside a transaction block that automatically commits a bundle, then you must commit the bundle yourself. To do so, add `bundle.commit` to your code.

## Entity Builder Examples

The following examples illustrate various ways that you can use builders to create sample data for use in GUnit tests.

- Creating Multiple Objects from a Single Builder

- Nesting Builders
- Overriding Default Builder Properties
- Creating a Producer with an Existing Commission Plan
- Creating an Account and a PolicyPeriod using PaymentPlan from a Builder
- Using Builders in Multiple Places

## 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 argument. For example, suppose that you want to create an Account that has a Policy. In this situation, you might want to do the following:

```
Account account = new AccountBuilder()
    .withPolicies(new PolicyBuilder().withDefaultPolicyPeriod())
    .create()
```

## Overriding Default Builder Properties

The following code samples illustrates multiple ways to create an Account object. The first code sample shows a simple test method and uses a transaction block. The `Transaction` object takes a block, which assigns the new account to the variable in the scope outside of the transaction.

```
function myTest(){
    var account : Account
    Transaction.runWithNewBundle( \ bundle -> {
        account = new AccountBuilder().create(bundle)
    })
}
```

There are generally two kinds of accounts: person and company. By default, `AccountBuilder` creates a person account. If you want a company account, then you need to assign a company contact as the account holder, as shown in the following code sample:

```
account = new AccountBuilder(false)
    .withAccountHolderContact(new PolicyCompanyBuilder(42))
    .create(bundle)
}
```

In this example, passing `false` to `AccountBuilder` tells it not to create a default account holder. Instead, you pass in your own account holder by calling `withAccountHolderContact`, which takes a `ContactBuilder`. In this case, `PolicyCompanyBuilder` suffices. The passed in number 42 seeds the default data with something unique (ideally) and identifiable.

The following example creates a company account and overrides some of the default values. Anywhere you see `code`, it means numerical seed value. (String variants derive from the given values.) It also illustrates how to nest the results of one builder inside another.

```
var address = new AddressBuilder()
    .withAddressLine1( codeStr + " Main St." )
    .withAddressLine2( "Suite " + codeStr )
    .withCity( "San Mateo" )
    .withState( "CA" )
    .withPostalCode( "94404-" + codeStr )
```

```

    .asBusinessType()

var company = new PolicyCompanyBuilder(code, false)
    .withCompanyName( "This Company " + code )
    .withWorkPhone( "650-555-" + codeStr )
    .withAddress(address)
    .withOfficialID( new OfficialIDBuilder().withType( "FEIN" ).withValue( "11-222" + codeStr ) )

var account = new AccountBuilder(false)
    .withIndustryCode("1011", "SIC")
    .withAccountOrgType( "Corporation" )
    .withAccountHolderContact(company)
    .create(bundle)

```

The following example takes the previous code and presents it as a single builder that takes other builders as arguments. While more compact, it also takes more planning and understanding of builders to create. Notice the successive levels of indenting used to signal the creation of a new (embedded) builder.

```

var account = new AccountBuilder(false)
    .withIndustryCode("1011", "SIC")
    .withAccountOrgType( "Corporation" )
    .withAccountHolderContact(new PolicyCompanyBuilder(code, false)
        .withCompanyName( "This Company " + code )
        .withWorkPhone( "650-555-" + codeStr )
        .withAddress( new AddressBuilder()
            .withAddressLine1( codeStr + " Main St." )
            .withAddressLine2( "Suite " + codeStr )
            .withCity( "San Mateo" )
            .withState( "CA" )
            .withPostalCode( "94404-" + codeStr )
            .asBusinessType() )
        .withOfficialID( new OfficialIDBuilder()
            .withType( "FEIN" )
            .withValue( "11-222" + codeStr ) )
    )
    .create(bundle)

```

## Creating a Producer with an Existing Commission Plan

The following code sample uses multiple builders to create a producer with a commission plan. It also creates an account with a policy that has that producer as the primary producer.

```

var randomNumber = gw.api.databuilder.SequentialIntegerGenerator
var currentDate = gw.api.util.DateUtil.currentDate()
var accountId = "DEV_AgencyBillAccount-" + randomNumber
var agencyBillPolicyNumber = "DEV_AgencyBillPolicy_-" + randomNumber
var producerName = "DEV_AgencyBillProducer_-" + randomNumber

var producer = new ProducerBuilder()
    .withName( producerName )
    .withProducerCodeHavingCommissionPlan( "PC-" + randomNumber, "QA1COMMISSIONPLAN01" )
    .create()

var producerCode : ProducerCode = producer.ProducerCodes[0]

var account = new AccountBuilder()
    .asMediumBusiness()
    .withBillingPlan( "QA1BILLINGPLAN01" )
    .withDelinquencyPlan( "QA1DELINQUENCYPLAN01" )
    .create()

var policyPeriod = new PolicyPeriodBuilder()
    .onAccount( account )
    .withProductCpp()
    .asAgencyBill()
    .withPrimaryProducerCode( producerCode )
    .withPolicyNumber( agencyBillPolicyNumber )
    .withPaymentPlan( "QA1PAYMENTPLAN01" )
    .withPremiumWithDepositAndInstallments( 21000 )
    .createAndCommit()

```

## Creating an Account and a PolicyPeriod using PaymentPlan from a Builder

The following code sample creates an Account and a PolicyPeriod using a Payment Plan builder.

```

var paymentPlan = new PaymentPlanBuilder()
    .withName("10 Down, 11 Payments, Monthly")
    .monthly()

```

```

.withDeposit(10)
.withNumPayments(11)
.withLatestLastInvoiceSent(60)
.createAndCommit()

var account = new AccountBuilder()
.named("Sample Account Name")
.withNumber("ACCT_1001")
.withBillingPlan("QA1BILLINGPLAN01")
.withDelinquencyPlan("QA1DELINQUENCYPLAN01")
.withDistributionUpToAmountUnderContract()
.withInvoiceDayOfMonth(1)
.createAndCommit()

var policyPeriod = new PolicyPeriodBuilder()
.withPolicyNumber("PolicyNumber001")
.onAccount(account)
.withPaymentPlan(paymentPlan)
.withOneYearPeriodStartingOn("2008-03-04")
.withPremiumWithDepositAndInstallments(1000)
.createAndCommit()

```

## Using Builders in Multiple Places

Data builders use separate bundles from the BillingCenter interface and batch processes. If you run interface actions between builders, the bundle goes stale. To solve this you need to reset the entity bundle.

```

var account = new AccountBuilder()
.named("someAccountName")
.withNumber("someAccountNumber")
.asCreditCard()
.create()

var requirement = new CollateralRequirementBuilder()
.onAccount(account)
.withRequired(300)
.asLetterOfCredit()
.withName("initialRequirementName")
.createAndCommit()

//various BillingCenter actions that make the Builder Bundle stale
TestFixtureBuilder.resetEntityBundle()
var bundle = TestFixtureBuilder.getEntityBundle()

new CollateralRequirementBuilder()
.onAccount(account)
.withRequired(100)
.asLetterOfCredit()
.withName("newRequirementName")
.createAndCommit()

```

## Creating New Builders

If you need additional builder functionality than that provided by the BillingCenter 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 405
- “Extending the DataBuilder Class” on page 405
- “Creating a Builder for a Custom Entity and Testing It” on page 407

## Extending an Existing Builder Class

To extend an existing builder class, use the following syntax:

```
class MyExtendedBuilder extends SomeExistingBuilder {  
    construct() {  
        ...  
    }  
    ...  
    function someNewFunction() : MyExtendedBuilder {  
        ...  
        return this  
    }  
    ...  
}
```

The following `MyPersonBuilder` class extends the existing `PersonBuilder` class. The existing `PersonBuilder` class contains methods to set both the first and last names of the person, but not the person's middle name. The new extended class contains a single method to set the person's middle name. As there is no static field for the properties on a type, you must look up the property by name.

```
uses gw.api.databuilder.PersonBuilder  
  
class MyPersonBuilder extends PersonBuilder {  
  
    construct() {  
        super( true )  
    }  
  
    function withMiddleName( testname : String ) : MyPersonBuilder {  
        set(Person.TypeInfo.getProperty( "MiddleName" ), testname)  
        return this  
    }  
}
```

The `PersonBuilder` class has two constructors. This code sample uses the one that takes a Boolean that means create this class `withDefaultOfficialID`.

Another more slightly complex example would be if you extended the `Person` object and added a new `PreferredName` property. In this case, you might want to extend the `PersonBuilder` class also and add a `withPreferredName` method to populate that field through a builder.

## Extending the DataBuilder Class

To extend the `DataBuilder` class, use the following syntax:

```
class MyNewBuilder extends DataBuilder<BuilderEntity, BuilderType> {  
    ...  
}
```

The `DataBuilder` class takes the following parameters:

Parameter	Description
<code>BuilderEntity</code>	Type of entity created by the builder. The <code>create</code> method requires this parameter so that it can return a strongly-typed value and, so that other builder methods can declare strongly-typed parameters.
<code>BuilderType</code>	Type of the builder itself. The <code>with</code> methods require this on the <code>DataBuilder</code> class so that it can return a strongly-typed builder value (to facilitate the chaining of <code>with</code> methods).

If you choose to extend the `DataBuilder` class (`gw.api.databuilder.DataBuilder`), place your newly created builder class in the `gw.api.databuilder` package in the Studio Tests folder. Start any method that you define in your new builder with one of the recommended words (described previously in “Creating an Entity Builder” on page 399):

Initial word	Indicates
on	A link to a parent, for example, <code>PolicyPeriod</code> is on an <code>Account</code> , so the method is <code>onAccount(Account account)</code> .
as	A property that holds only a single state, for example: <code>asBusinessType</code> or <code>as AgencyBill</code> .
with	The single element or property to be set. For example, the following sets a <code>FirstName</code> property: <code>withFirstName("Joe")</code>

Your configuration methods can set properties by calling `DataBuilder.set` and `DataBuilder.addArrayElement`. You can provide property values as any of the following:

- Simple values.
- Beans to be used as subobjects.
- Other builders, which BillingCenter 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 BillingCenter configures that property on the target object. (BillingCenter 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. BillingCenter processes properties with the same order value (for example, all those that do not have an order) in the order in which they are set on the builder.

In most cases, Guidewire recommends that you omit the order value as you are implement builder configuration methods. This enables callers of your builder to select the execution order through the order of the configuration method calls.

Constructors for builders can call `set`, and similar methods to set up default values. These are useful to satisfy `null` constraints so it is possible to commit built objects to the database. However, Guidewire generally recommends that you limit the number of defaults. This is so that you have the maximum control over the target object.

### Other DataBuilder Classes

The `gw.api.databuilder` package also includes `gw.api.databuilder.ValueGenerator`. You can use this class, for example, to generate a different value for each instance constructed to satisfy uniqueness constraints. The `databuilder` package includes `ValueGenerator` class variants for generating unique integers, strings, and type-keys:

- `gw.api.databuilder.IntegerStringGenerator`
- `gw.api.databuilder.SequentialStringGenerator`
- `gw.api.databuilderTypekeyStringGenerator`

### Custom Builder Populators

Ideally, all building can be done through simple property setters, using the `DataBuilder.set` or `DataBuilder.addArrayElements` methods. However, you may want to define more complex logic, if these methods do not suffice. To achieve this, you can define a custom implementation of `gw.api.databuilder.populator.BeanPopulator` and pass it to `DataBuilder.addPopulator`. Guidewire provides an abstract implementation, `AbstractBeanPopulator`, to support short anonymous `BeanPopulator` objects.

The following example uses an anonymous subclass of `AbstractBeanPopulator` to call the `withCustomSetting` method. This code passes the group to the constructor, and the code inside of `execute` only accesses it through the `vals` argument. This allows the super-class to handle packaging details.

```
public MyEntityBuilder withCustomSetting( group : Group ) {
    addPopulator( new AbstractBeanPopulator<MyEntity>( group ) {
        function execute( e : MyEntity, vals : Object[] ) {
            e.customGroupSet( vals[0] as Group )
        }
    })
    return this
}
```

The `AbstractBeanPopulator` class automatically converts builders to beans. That is, if you pass a builder to the constructor of `AbstractBeanPopulator`, it returns the bean that it builds in the `execute` method. The following example illustrates this.

```
public MyEntityBuilder withCustomSetting( groupBuilder : DataBuilder<Group, ?> ) : MyEntityBuilder {
    addPopulator( new AbstractBeanPopulator<MyEntity>( groupBuilder ) {
        function execute( e : MyEntity, vals : Object[] ) {
            e.customGroupSet( vals[0] as Group )
        }
    })
    return this
}
```

### [Creating a Builder for a Custom Entity and Testing It](#)

It is also possible, if you want, to create a builder for a custom entity. For example, suppose that you want each BillingCenter user to have an array of external credentials (for automatic sign-on to linked external systems, perhaps). To implement, you can create an array of `ExtCredential` on `User`, with each `ExtCredential` having the following parameters:

Parameter	Type
ExtSystem	Typekey
UserName	String
Password	String

After creating your custom entity and its builder class, you would probably want to test it. To accomplish this, you need to do the following:

Task	Affected files	See
1. Create a custom <code>ExtCredential</code> array entity and extend the <code>User</code> entity to include it.	<code>ExtCredential.eti</code> <code>User.etcx</code>	To create a custom entity
2. Create an <code>ExtCredentialBuilder</code> by extending the <code>DataBuilder</code> class and adding <code>withXXX</code> methods to it.	<code>ExtCredentialBuilder.gs</code>	To create an <code>ExtCredentialBuilder</code> class
3. Create a test class to exercise and test your new builder.	<code>ExtCredentialBuilderTest.gs</code>	To create an <code>ExtCredentialBuilderTest</code> class

#### **To create a custom entity**

To create a new array `ExtCredential` custom entity, you need to do the following:

- Add the `ExtSystem` typelist (in the `Typelist` editor in Guidewire Studio).

- Define the ExtCredential array entity (in ExtCredential.eti, accessible through Guidewire Studio).
  - Modify the array entity definition to include a foreign key to User (in ExtCredential.eti).
  - Add an array field to the User entity (in User.etx).
1. Add an ExtSystem typelist. Within Guidewire Studio, navigate to **Typelist**, and then right-click **New → Typelist**. Add a few *external system* typecodes. (For example, add SystemOne, SystemTwo, or similar items.)
  2. Create ExtCredential. Right-click **Entity**, and then click **New → Entity**. Name this file ExtCredential.eti and enter the following:

```
<?xml version="1.0"?>
<entity xmlns="http://guidewire.com/datamodel" entity="ExtCredential" table="extcred"
    type="retireable" exportable="true" platerform="true" >
    <typekey name="ExtSystem" typelist="ExtSystemType" desc="Type of external system"/>
    <column name="UserName" type="shorttext"/>
    <column name="Password" type="shorttext"/>
    <foreignkey name="UserID" fkentity="User" desc="FK back to User"/>
</entity>
```

3. Modify the User entity. Find User.etx (in **Extensions → Entity**). If it does not exist, then you must create it. However, most likely, this file exists. Open the file and add the following:

```
<array name="ExtCredentialRetirable" arrayentity="ExtCredential"
    desc="An array of ExtCredential objects" arrayfield="UserID" exportable="false"/>
```

See “Extending a Base Configuration Entity” on page 209 for information on extending the Guidewire BillingCenter 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 405 for a discussion of these two parameters.)
- It uses a constructor for the super class—`DataBuilder`—that requires the entity type to create.
- It implements multiple `withXXX` methods that populate an `ExtCredential` array object with the passed in values.

### To create an ExtCredentialBuilderTest class

Finally, to be useful, you need to reference your new builder in Gosu code. You can, for example, create a GUnit test that uses the `ExtCredentialBuilder` class to create test data. Place this class in its own package in the `Tests` folder.

```
package MyTests

uses AllMyClasses.ExtCredentialBuilder
uses gw.transaction.Transaction

@gw.testharness.ServerTest
class ExtCredentialBuilderTest extends gw.testharness.TestBase {

    static var credential : ExtCredential
    construct() {

    }

    function beforeClass () {
        Transaction.runWithNewBundle( \bundle -> {
            credential = new ExtCredentialBuilder()
                .withType( "SystemOne" )
                .withUserName( "Peter Rabbit" )
                .withPassword( "carrots" )
                .create( bundle )
        })
    }

    function testUsername() {
        assertEquals("User names do not match.", credential.UserName, "Peter Rabbit")
    }

    function testPassword() {
        assertEquals("Passwords do not match.", credential.Password, "carrots")
    }
}
```

Notice the following about this code sample:

- It includes the `uses` statements for both `ExtCredentialBuilder` and `gw.transaction.Transaction`.
- It creates a static `credential` variable. As the code declares this variable outside of a method—as a class variable—it is available to all methods within the class. (GUnit maintains a single copy of this variable.) As you run a test, GUnit creates a single instance of the test class that each test method uses. Therefore, to preserve a variable value across multiple test methods, you must declare it as a static variable. (For a description of the `static` keyword and how to use it in Gosu, see “Static Modifier” on page 212 in the *Gosu Reference Guide*.)
- It uses a `beforeClass` method to create the `ExtCredential` test data. This method calls `ExtCredentialBuilder` as part of a transaction block, which creates and commits the bundle automatically. GUnit calls the `beforeClass` method before it instantiates the test class for the first time. Thereafter, the test class uses the test data created by the `beforeClass` method. It is important to understand that GUnit does not drop the database between execution of each test method within a test class. However, if you run multiple test classes together (for example, by running all the test classes in a package), GUnit resets the database between execution of each test class.
- It defines several test methods, each of which starts with `test`, with each method including an `assertEquals` method to test the data.

If you run the `ExtCredentialBuilderTest` class as defined, the GUnit tester displays green icons, indicating that the tests were successful:



# Guidewire BillingCenter Configuration



# BillingCenter Workflows and Delinquency Plans

This topic covers the interaction between BillingCenter workflow and BillingCenter delinquency plans. For general information on workflow, see “Guidewire Workflow” on page 341.

This topic includes:

- “Workflows in BillingCenter” on page 413
- “BillingCenter Workflows and Delinquency Events” on page 414
- “Creating a New Delinquency Plan: An Example” on page 418

## Workflows in BillingCenter

All BillingCenter workflows are subtypes of the `Workflow` object. In the base configuration, Guidewire subtypes the `Workflow` object as `BCWorkflow`. BillingCenter subtypes `BCWorkflow` as `AgencyBillWorkflow` and `DelProcessWorkflow`. BillingCenter then subtypes these two workflows. This creates the following workflow subtypes in Studio:

Workflow	Subtypes
<code>AgencyBillWorkflow</code>	<code>LegacyAgencyBill</code> <code>StdAgencyBill</code>
<code>DelProcessWorkflow</code>	<code>CancelImmediately</code> <code>LegacyDelinquency</code> <code>LegacyDelinquencyOther</code> <code>SimpleDelinquency</code> <code>SimpleFailureToReport</code> <code>SimpleProducerRefer</code> <code>StdDelinquency</code>

For more information on the `Workflow` object, see the *BillingCenter Data Dictionary*.

BillingCenter specifies a set of reasons for why the delinquency process started in the `DelinquencyReason` type-list. In the base configuration, BillingCenter associates a delinquency reason with one or more delinquency plans. (There can be multiple plans as the reason for the delinquency is the same, but the plan for how to handle it is different.) BillingCenter also provides a base configuration workflow for many of the base configuration delinquency reasons.

The following table describes this relationship:

Delinquency reason	Workflow
Failure to report	<code>SimpleFailureToReport</code>
Other	<code>LegacyDelinquencyOther</code>
Past due	<code>StdDelinquency</code> <code>LegacyDelinquency</code>
Producer referred	<code>SimpleProducerRefer</code>

## BillingCenter Workflows and Delinquency Events

It is possible, in BillingCenter, to associate a delinquency event with a workflow step programmatically. It is important to understand that a delinquency event is not the same type of event as an entity event that you can define through the workflow `Events` tab. BillingCenter defines specific entity events that it associates with an entity. For example, BillingCenter associates the following events with the `Account` object:

- `AccountAdded`
- `AccountChanged`
- `AccountRemoved`

For a list of the events that BillingCenter associates with a specific object, see “List of Messaging Events in BillingCenter” on page 323 in the *Integration Guide*.

You can associate one or more delinquency events with a workflow step. However, there is not necessarily a one-to-one mapping. A *delinquency event* is a description of an action performed by a workflow step that you want to communicate to the BillingCenter user.

BillingCenter defines specific delinquency events in the `DelinquencyEventName` type list. The following table lists the delinquency events. Notice that:

- You use the `Code` value in Gosu.
- You see the `Name` value within BillingCenter (as the delinquency event).

Code	Name	Description
<code>Canceled</code>	Cancellation Received	Cancellation Billing Instruction Received
<code>CancelInPAS</code>	Cancellation Request	Send Cancellation Request to PAS
<code>Cancellation</code>	Notice of Intent To Cancel	Send Notice of Intent To Cancel
<code>Collections</code>	Collections	Collections
<code>DunningLetter1</code>	Dunning Letter 1	Dunning Letter 1
<code>DunningLetter2</code>	Dunning Letter 2	Dunning Letter 2
<code>LateFee</code>	Late Fee	Late Fee
<code>RescindReinstate</code>	Rescind/Reinstate	Rescind/Reinstate
<code>Writeoff</code>	Writeoff	Writeoff

## Delinquency Events in Studio

Within Studio, you can associate a delinquency event with a workflow step through the use of Gosu code, usually in the `Enter Script` block for the workflow step. For example, the `FirstDunningLetter` workflow step in the `StdDelinquency` workflow contains the following code in the `Enter Script` block:

```
// send first dunning letter
dlnqProcess.Target.sendDunningLetter()
dlnqProcess.flagEventCompleted(typekey.DelinquencyEventName.TC_DUNNINGLETTER1)

// charge policy late fee
dlnqProcess.chargeLateFee()
dlnqProcess.flagEventCompleted(typekey.DelinquencyEventName.TC_LATEFEE)
```

Notice the following about this code:

- It contains code related to two different delinquency events: `DunningLetter1` and `LateFee`.
- It sets the `DunningLetter1` delinquency event to completed. (There can be many other different steps associated a dunning letter, such as sending, approving, or cancelling.)
- It calls a method to create a late fee for the delinquent user and sets the `LateFee` delinquency event to completed. (There is no specific workflow step labeled Late Fee in the base configuration `StdDelinquency` workflow.)
- The completion of these two delinquency events signals that the workflow is ready to move to the next step.

## Delinquency Events in BillingCenter

Within BillingCenter, you need to define one event in your delinquency plan for each delinquency event that your code executes in your workflow in Studio. For example, the base configuration `LegacyDelinquency` workflow contains the following steps and associated delinquency events. (The BillingCenter delinquency event names come from the `DelinquencyEventName` typelist.)

Workflow step	Associated Gosu event	Delinquency event
<code>FirstDunningLetter</code>	<code>DUNNINGLETTER1</code>	Dunning Letter 1
<code>SecondDunningLetter</code>	<code>DUNNINGLETTER2</code>	Dunning Letter 2
<code>Cancellation</code>	<code>CANCELLATION</code>	Notice of Intent To Cancel
<code>ExitDelinquencyAfterCancellation</code>	<code>RESCINDREINSTATE</code>	Rescind/Reinstate

In some cases, the workflow contains several events associated with a single step. For example, in the base configuration `StdDelinquency` workflow, the `FirstDunningLetter` step contains multiple associated delinquency events:

Workflow step	Associated Gosu event	Delinquency event
<code>FirstDunningLetter</code>	<code>DUNNINGLETTER1</code> <code>LATEFEE</code>	Dunning Letter 1 Late Fee

As you create a delinquency plan in BillingCenter, you must associate the correct workflow delinquency events with the events in your delinquency plan. If you have multiple delinquency events on a single workflow step, then you need to set the relative order of occurrence within the delinquency plan.

You add delinquency events to a delinquency plan through the `Workflow` tab of the delinquency plan. For details, see “Adding Delinquency Events to a BillingCenter Delinquency Plan” on page 417.

### To view delinquency events within BillingCenter

1. Log into BillingCenter using an administrative account.
2. Navigate to `Administration` → `Delinquency Plans`. BillingCenter opens a list of delinquency plans in the center pane.

3. Select a delinquency plan and open the **Workflow** tab.

## BillingCenter Delinquency Event Fields

Within BillingCenter, if you navigate to the **Workflow** tab of a delinquency plan, you see a number of event-related fields. These delinquency events provide visual feedback on the status of the delinquency for an account or policy period.

For example, if you select the **Standard Delinquency** plan and open the **Workflow** tab, you see the following if you select **Past Due**:

The screenshot shows the 'Standard Delinquency Plan' configuration screen. At the top, there are 'Edit' and 'Clone' buttons. Below them, two tabs are visible: 'General' (selected) and 'Workflow'. In the 'Workflow' tab, there is a table with three columns: 'Delinquency Reason', 'Workflow Type', and 'Events'. Two rows are present: 'Failure to Report' (Workflow Type: Simple Failure to Report) and 'Past Due' (Workflow Type: Standard Delinquency). The 'Events' section at the bottom contains a table with columns: 'Event Name', 'Trigger', 'Offset', 'Relative Order', and 'Automatic'. Six events are listed: Dunning Letter 1, Late Fee, Dunning Letter 2, Notice of Intent To Cancel, Collections, Writeoff, and Rescind/Reinstate.

Delinquency Reason	Workflow Type	Events
Failure to Report	Simple Failure to Report	
Past Due	Standard Delinquency	Dunning Letter 1, Late Fee, Dunning Letter 2, Notice of Intent To Cancel, Collections, Writeoff, Rescind/Reinstate

Events				
Event Name	Trigger	Offset	Relative Order	Automatic
Dunning Letter 1	Inception Date	0	0	Yes
Late Fee	Inception Date	0	1	Yes
Dunning Letter 2	Inception Date	7		Yes
Notice of Intent To Cancel	Inception Date	14		Yes
Collections	Inception Date	45		No
Writeoff	Inception Date	120		No
Rescind/Reinstate	Good Standing			Yes

These fields have the following meanings, in general. However, they are dependent on the workflow implementation. These definitions assume the base configuration StdDelinquency workflow. Other workflow types can (and do) choose to implement the trigger dates differently, for example.

- **Delinquency Reason** – The initial reason for starting the delinquency process. In BillingCenter, you associate delinquency plans with accounts. As you set up the account, you specify the associated delinquency plan on the **Account Summary** screen. You specify the reason for the delinquency on the **Start Delinquency** screen off the **Policy Summary** screen for the associated policy. You must associate a workflow type with a delinquency reason. BillingCenter defines the set of delinquency reasons in the **DelinquencyReason** typelist.
- **Workflow Type** – The type of workflow associated with this particular delinquency reason. BillingCenter defines the set of workflows in the **Workflow** typelist, which is extendable.
- **Events** – The list of delinquency events associated with this workflow type. There are more details at the bottom of the screen. If you do not see any delinquency events, then click the arrow next to a delinquency reason to expand it. Not all delinquency reasons have associated delinquency events.
- **Event Name** – The name of the event. BillingCenter defines a set of delinquency-related events in the base configuration in the **DelinquencyEventName** typelist. You can extend this typelist.
- **Trigger** – Do not confuse the word, trigger, with the name of the TRIGGER branch. They are related. However, the discussion of workflow uses both in discussing the advancing of a workflow from one step to the next. In this context, BillingCenter uses this value as the basis for calculating the exact point at which the timeout for a delinquency event occurs.

Guidewire defines a set of delinquency-related triggers in the `DelinquencyTriggerBasis` typelist. You can extend this typelist, also. (You cannot alter values set in gray, however.) In the base configuration, this typelist contains the following defined triggers:

Code	Description
Canceled	Policy canceled
External	Externally-triggered event
Goodstanding	Account returned to good standing.
Inception	Occurs as the workflow calls the <code>DelinquencyProcess.inception</code> method, which the <code>StdDelinquency</code> workflow invokes after the grace period using the <code>onInception</code> method. However, this can differ in the other workflows.
PaidThroughDate	See the <code>PaidThroughDate</code> property on the <code>PolicyPeriod</code> in the <i>BillingCenter Data Dictionary</i> .

- **Offset** – BillingCenter uses this value to calculate the target date for the delinquency event. After BillingCenter creates the delinquency, you see this date on the `Delinquencies` page for each event. This value can be any positive or negative integer.
- **Relative Order** – The order in which delinquency events execute if a workflow step contains multiple delinquency events with the same target date (meaning the trigger basis plus the offset).
- **Automatic** – If Yes, the workflow does not require any external or outside action to move to the next step. If No, the workflow does the following:
  - It calls the `createApprovalActivity` method on the `DelinquencyProcessExt` extension class, which automatically generates an approval activity.
  - It enters a timeout step configured for a one hour wait (in the base configuration for `StdDelinquency`). After the timeout completes, the workflow checks the `Approved` status for the activity.
  - If the activity is not yet approved, the workflow continues to check the approval status of the activity periodically (as determined by the timeout).
  - After a user manually approves the activity, the `Approved` status for the activity becomes true. This makes the condition on the branch to the next step true. The workflow then takes that branch and continues to the next step.

## Adding Delinquency Events to a BillingCenter Delinquency Plan

You can add delinquency events to an existing delinquency plan (by editing it) or by cloning the plan and modifying the delinquency events associated with a cloned plan.

**Note:** The trigger and offset values that you set through the BillingCenter `Workflow` screen override any default values that you set for the workflow through Guidewire Studio. The default values in the workflow are based on these offsets. Therefore, changing the offset values in BillingCenter changes the workflow. (You may have to restart to see the changes.)

### To add delinquency events to an existing workflow

1. Log into BillingCenter using an administrative account.
2. Navigate to `Administration` → `Delinquency Plans`. BillingCenter opens a list of delinquency plans in the center pane.
3. Select a delinquency plan and click either `Edit` or `Clone`.
  - If you choose `Edit`, you can only modify a subset of the plan parameters. You can, however, add additional delinquency events to the plan in the `Workflow` tab.
  - If you choose `Clone`, BillingCenter opens a new screen, with two tabs, that you can use to specify various parameters on the cloned workflow. Again, you can use the `Workflow` tab to modify the delinquency events associated with the cloned delinquency plan.

## Creating a New Delinquency Plan: An Example

Creating a new delinquency plan is a multi-step process:

1. Create a workflow process in Guidewire Studio. (This can be one that exists in the base configuration, one that you modify, or one that you newly create.)
2. Associate specific delinquency events with specific workflow steps in Guidewire Studio. (Typelist `DelinquencyEventName` lists the valid delinquency events.)
3. Create a new delinquency plan in BillingCenter.
4. Add delinquency events in BillingCenter and associate each one with the delinquency event that you set up in your delinquency workflow step in Studio.

The following sections illustrate this process.

**Note:** It is not sufficient to add or remove delinquency plan events through the BillingCenter interface. You must also make the corresponding changes to the associated workflow in Studio for these changes to take effect.

### To create a delinquency workflow in Studio

1. Select the `Workflows` node, right-click and select **Create metadata for a new workflow subtype**.
2. In the **New Workflows Subtype Metadata** dialog, enter the following:

<b>Entity</b>	MyStdDelinquency
<b>Supertype</b>	<code>DelProcessWorkflow</code>
<b>Description</b>	My Delinquency Workflow

3. After entering this text, click **Gen to clipboard**. This action generates the necessary XML code that you need to create the new workflow subtype.
4. Navigate to **Data Model Extensions → extensions** and select **New → Other file** using the right-click menu.
5. Enter `MyStdDelinquency.eti` for the file name.
6. Populate this file with the following code (from the clipboard):
 

```
<?xml version="1.0"?>
<subtype desc="My Standard Delinquency" entity="MyStdDelinquency" supertype="DelProcessWorkflow"/>
```
7. Close and restart Studio. (You also need to restart the application server, if it is currently running.) You now see a `MyStdDelinquency` entry added to the `Workflow` typelist and a new `MyStdDelinquency` workflow type added to `Workflows` in the `Resources` tree.
8. Select `MyStdDelinquency` from `Workflows` in the `Resources` tree.
9. Right-click and select **New → Workflow Process** from the menu. Studio opens an **Outline** view (in the center pane) and a layout workspace for the new workflow process (in the right-hand pane). Initially:
  - The outline view contains the few required workflow elements.
  - The layout view contains a default outcome (`DefaultOutcome`).
10. Select `<Context>` in the outline view. Define the following symbol:
  - **Name:** `dlnqProcess`
  - **Type:** `DelinquencyProcess`
  - **Value:** `Workflow.DelinquencyProcess`

You need this symbol for the following steps. (For information on symbols, see “`<Context>`” on page 346.)

**11.** Select the **DefaultOutcome** step.

a. Change its name to **Canceled**. (Right-click and choose **Rename**.)

b. Enter the following in the **Enter Script** block in the properties area at the bottom of the screen:

```
// cancel target
dInqProcess.cancelTarget()
dInqProcess.Target.CancelStatus = PolicyCancelStatus.TC_CANCELED
dInqProcess.flagEventCompleted(typekey.DelinquencyEventName.TC_CANCELED)
```

**12.** Create a **FirstDunningLetter** step. Right-click in an empty layout workspace area and select **New ManualStep**.

Enter the following:

- **Step ID:** FirstDunningLetter
- **Type:** TRIGGER
- **ID:** Cancel

Click **OK** to create the step, then enter the following in the **Enter Script** block in the properties area at the bottom of the screen:

```
// send first dunning letter
dInqProcess.Target.sendDunningLetter()
dInqProcess.flagEventCompleted(typekey.DelinquencyEventName.TC_DUNNINGLETTER1)

// charge policy late fee
dInqProcess.chargeLateFee()
dInqProcess.flagEventCompleted(typekey.DelinquencyEventName.TC_LATEFEE)
```

**13.** Create an **Inception** step (a manual step). Enter the following:

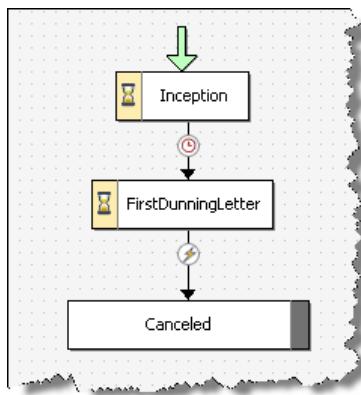
- **Step ID:** Inception
- **Type:** Timeout
- **To:** FirstDunningLetter
- **Timeout Absolute:**  
`dInqProcess.getApprovalDate(typekey.DelinquencyEventName.TC_DUNNINGLETTER1)`

Click **OK** to create the step, then enter the following in the **Enter Script** block in the properties area at the bottom of the screen:

```
// Set the process inception date and fire off history events
dInqProcess.onInception()
```

**14.** Select **<Start>** in the outline view. Change **First Step** to **Inception**.

At this point, you have a very simple delinquency workflow that looks similar to the following:



**To associate a delinquency plan with workflow delinquency events**

1. Log into BillingCenter using an administrative account.
2. Open the **Administration** tab.

3. Select **New Delinquency Plan** from the **Actions** menu. BillingCenter opens a new **Delinquency Plan** screen, with two tabs, that you can use to specify the various parameters on the delinquency plan.
4. In the **General** tab, enter the required fields. For example, enter **My Delinquency Plan** for **Name**, then choose appropriate values for the other required fields.

**General**

**Delinquency Plan**

- Name: \* My Delinquency Plan
- Effective Date: \* 04/09/2009
- Expiration Date: ..../.....

**Cancellation Target**

- Cancellation Target: \*  All Policies in Account  Delinquent Policy Only
- Hold Invoicing: \*  Yes  No
- on Targeted Policies

**Grace Period**

- Grace Period: \* 0 Days

**Fees**

- Late Fee: \* \$100.00
- Reinstatement Fee: \* \$25.00

**Amount Thresholds**

- Writeoff Threshold: \* \$25.00
- Enter Delinquency Threshold (Account): \* \$35.00
- Enter Delinquency Threshold (Policy): \* \$50.00
- Cancel Delinquency Policy: \* \$75.00
- Exit Delinquency Policy: \* \$10.00

**Availability**

- Applicable Segments: Personal

5. In the **Workflow** tab, click **Add** to add a new reason. For example, select **Past Due**.
6. Select your newly created workflow for workflow type (**My Delinquency Workflow**, for example).
7. Under **Events**, click **Add**. BillingCenter opens a screen in which you can add delinquency events.

**Note:** In general, you need to define one event in your delinquency plan workflow for each event listed in your workflow in Studio.

8. Create a **Dunning Letter 1** event by entering the following (assuming that this is the **MyStdDelinquency** workflow):
  - **Name:** Dunning Letter 1
  - **Automatic:** Yes
  - **Offset:** 0
  - **Trigger:** Inception Date

9. Create a **Canceled** event by entering the following:
  - **Name:** Cancellation Received
  - **Automatic:** No
  - **Offset:** 10
  - **Trigger:** Inception Date

Setting **Automatic** to **No** indicates that the workflow requires approval of the cancellation before continuing.

After performing these steps, the **Workflow** tab looks similar to the following:

**My Delinquency Plan (Up to Delinquency Plans)**

**General** **Workflow**

**Events**

<input type="checkbox"/>	*Delinquency Reason	*Workflow Type	Events
<input checked="" type="checkbox"/>	Past Due	My Delinquency Workflow	Dunning Letter 1, Cancellation Request

Delinquency Reason \* Past Due | Workflow Type \* My Delinquency Workflow

<input type="checkbox"/>	*Event Name	*Trigger	Offset
<input type="checkbox"/>	Dunning Letter 1	Inception Date	0
<input type="checkbox"/>	Cancellation Request	Inception Date	10



# Configuring the Charge Invoicing Process

Charge invoicing is the process of dividing charges into invoice items and assigning the items to invoices. Charge invoicing occurs whenever a new charge is created or an existing charge is modified. For example, charge invoicing occurs when a change to a payment plan causes a recalculation of invoice items.

This topic includes:

- “The Charge Invoicing Process for New Charges” on page 423
- “Expected Results and Plugin Customizations” on page 427
- “Modifying an Existing Charge” on page 427
- “Extension Properties for Charge and InvoiceItem Objects” on page 429
- “Charge Invoicing and Payment Plans” on page 432
- “Payment Plan Modifiers” on page 433
- “InvoiceStream and DateSequence Plugins” on page 434

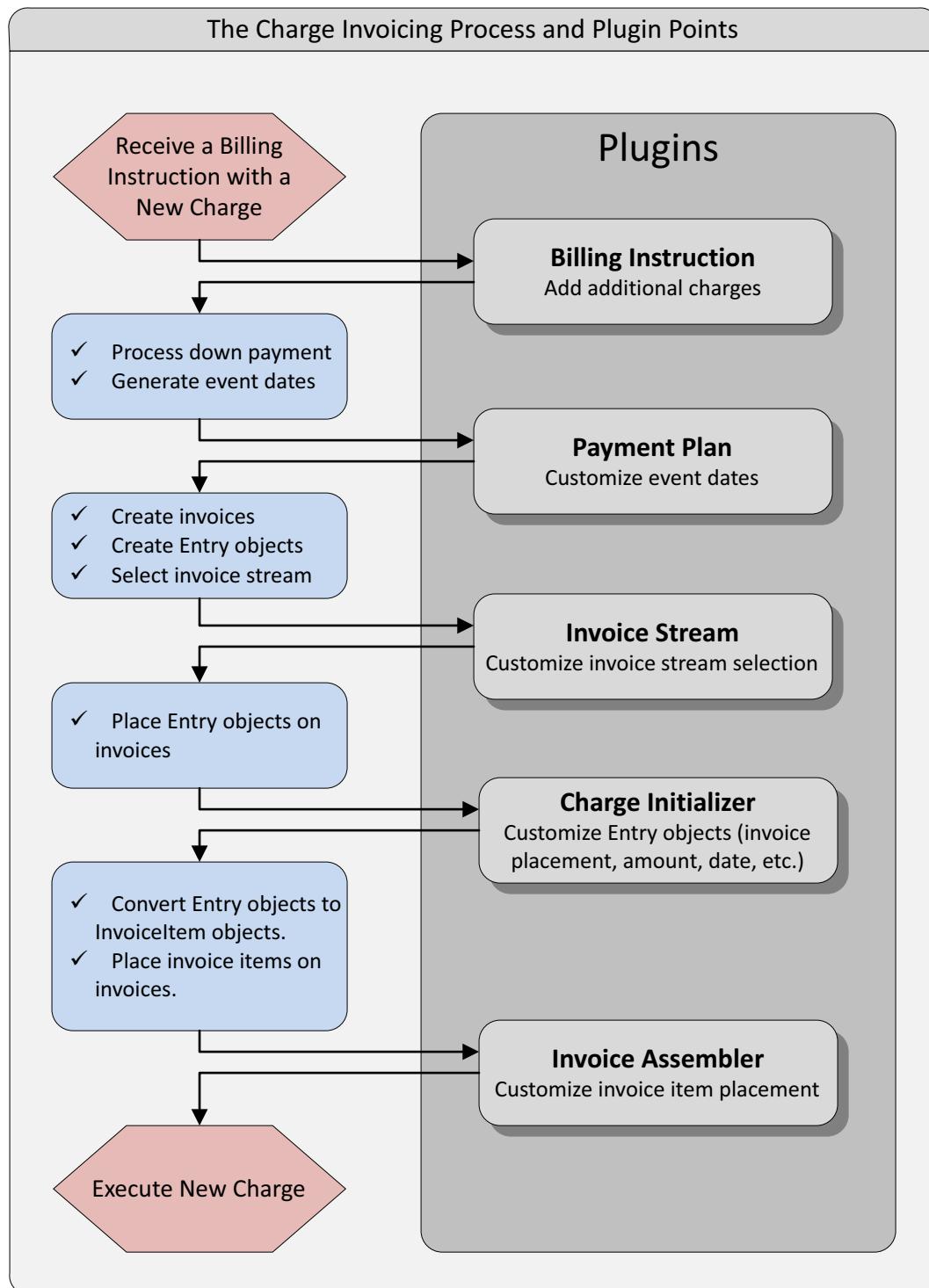
## The Charge Invoicing Process for New Charges

New charges typically arrive in the form of a billing instruction sent to BillingCenter from an external system. New charges can also be entered manually with the BillingCenter user interface. For information on entering new charges with the user interface, see “Charge Invoicing Process” on page 181 in the *Application Guide*.

During the charge invoicing process, BillingCenter performs several steps, including determining the date of each invoice item and the appropriate invoice to receive the item. The BillingCenter base configuration provides a complete implementation of the charge invoicing process. However, because invoicing requirements vary among carriers, the base configuration process can be configured to meet each carrier’s unique needs.

To enable configuration, BillingCenter invokes configurable plugin modules at various points in the charge invoicing process. Some plugins are empty placeholders intended to perform the special invoicing actions a particular carrier wishes to enact. Other plugins implement the invoicing actions of the base configuration. Through the use of the Gosu programming language, the plugins can be modified or extended to alter the base configuration's invoicing process.

The diagram shown below illustrates the steps of the charge invoicing process and the points at which plugins are invoked to enable customization of the process. The graphic includes several references to `Entry` objects. An `Entry` object is an invoice item under development. During the charge invoicing process, customization of an invoice item is performed upon its `Entry` object. At the conclusion of the process, each `Entry` object is ultimately converted to an `InvoiceItem` object having the same settings as the customized `Entry` object.



The following table provides a brief description of each plugin and links on where to find additional information about modifying the plugin.

Plugin	Description
Billing Instruction	Enables customization of the charge. See "Billing Instruction Execution Customization Plugin" on page 167 in the <i>Integration Guide</i> .
Payment Plan	Enables customization of event dates. See "Payment Plan Plugin" on page 174 in the <i>Integration Guide</i> .
Invoice Stream	Enables customization of the invoice stream selection. See "Invoice Stream Plugin" on page 180 in the <i>Integration Guide</i> .
Charge Initializer	Enables customization of invoice item amounts, dates, and the placement of invoice items on invoices. See "Using the ChargeInitializer Plugin" on page 172 in the <i>Integration Guide</i> .
Invoice Assembler	Enables customization of invoice item placement. See "Invoice Assembler Plugin" on page 177 in the <i>Integration Guide</i> .

## Expected Results and Plugin Customizations

When settings are defined or viewed in the BillingCenter user interface, expectations are created concerning the results of related operations. For example, when a policy is assigned a monthly payment plan, it is a reasonable expectation for invoices to be generated on a monthly basis. However, plugin customization can override and radically alter the expected result. As an extreme example, the Invoice Stream plugin can be customized to completely ignore the monthly setting and, instead, assign invoices based on another periodicity, such as weekly. To the BillingCenter user, such unexpected results can appear to be wild errors. Most customizations are not as extreme as this example, but customizations can alter expected results, typically in subtle ways.

The same type of unexpected result can occur when a plugin customization is altered by another plugin. For example, in the charge invoicing process, the Payment Plan plugin can customize event dates. Later in the process, the Charge Initializer plugin can modify the same event dates, potentially reversing the earlier customization.

Plugin customizations are performed behind the scenes, out of sight of the BillingCenter user. There is no procedure available that enables a user to view the customization that a particular plugin performs. Therefore, settings defined in BillingCenter are best viewed as guidelines that may be overridden by subsequent customizations.

## Modifying an Existing Charge

### The ChargeInstallmentChanger Class

Existing charges that have been executed and assigned to invoices can be modified with some restrictions. One restriction is that the original amount of the charge cannot be modified. The amounts of individual invoice items can be modified, new items created and existing items deleted or reversed. However, at the completion of all modifications, the sum of all the invoice items must still equal the original charge amount.

A second restriction is that multiple charges cannot be modified simultaneously or in tandem. Each individual charge must be performed and finished before beginning to modify another charge.

An existing charge is modified by using the `ChargeInstallmentChanger` class. The constructor for the class accepts an existing charge of the class type `Charge`.

The `ChargeInstallmentChanger` converts the invoice items into editable objects of the class type `Entry`. All modifications must be performed upon these `Entry` objects. New `Entry` objects can be created by calling the `ChargeInstallmentChanger.addEntry` method.

After completing all the desired modifications, the `ChargeInstallmentChanger.execute` method must be called to make the modifications permanent. Each `Entry` object of the `ChargeInstallmentChanger` is converted back to an invoice item assigned to a particular invoice. The modified charge becomes the charge of record and replaces the original invoicing arrangement.

To comply with the restriction against simultaneous charge modifications, a new `ChargeInstallmentChanger` instance must be finished by calling its `execute` method before creating another instance of the class. The required order of execution is demonstrated below.

```
/* Create changer1, perform the desired modifications, and finish by calling execute() */
ChargeInstallmentChanger changer1 = new ChargeInstallmentChanger( existingCharge )
/* ... Perform modifications to changer1 ... */
changer1.execute()

/* Now you can create changer2 and perform another modification */
ChargeInstallmentChanger changer2 = new ChargeInstallmentChanger( anotherExistingCharge )
/* ... Modifications to changer2 ... */
changer2.execute()
```

The following sample code provides examples on how to modify an existing charge using the `ChargeInstallmentChanger` class. Included are techniques for modifying the `Entry` objects. For details about the methods and properties used in the code sample, refer to the *Gosu API Reference*.

```
/** Modify an existing charge using ChargeInstallmentChanger ***/

/* Create a ChargeInstallmentChanger instance using an existing Charge object */
ChargeInstallmentChanger changer = new ChargeInstallmentChanger( existingCharge )

/* Retrieve the charge's invoice items in the form of a List of editable Entry objects */
List<ChargeInstallmentChanger.Entry> existingEntries = changer.getEntries()

/* Alternatively, given an InvoiceItem object, we can retrieve its editable Entry object */
ChangeInstallmentChanger.Entry entry = changer.getEntryFor( existingInvoiceItem )

/* With the ChargeInstallmentChanger object created, its Entry objects can be modified, removed,
 * or reversed, and new Entry objects can be created and added. The following code segments
 * demonstrate some of these operations.
 */

/* Add a new Entry to the ChargeInstallmentChanger object */
ChargeInstallmentChanger.Entry newEntry = changer.addEntry( $5,
    InvoiceItemType.TC_INSTALLMENT.get(),
    DateTimeUtil.getTodaysDate() )

/* Modify an Entry's amount */
newEntry.setAmount( $10 )

/* Before performing an Entry operation, such as removing or reversing it, methods are
 * provided to check whether the operation is valid on the Entry at the given moment.
 */
if( newEntry.canRemove() ){
    newEntry.remove()
}

/* Customer-defined extension properties for the InvoiceItem object can be set and retrieved from
 * the Entry object. The extension property values will be carried over to the InvoiceItem when the
 * Entry is converted.
 */
newEntry.setExtensionProperty( InvoiceItem#MyCommentProperty, "Special fee" )
String description = newEntry.getExtensionProperty( InvoiceItem#MyCommentProperty )

/* After modifications are complete, the ChargeInstallmentChanger object is executed, which
 * converts the Entry objects to InvoiceItem objects. InvoiceItem properties set on the Entry
 * object (such as the MyCommentProperty extension property set above) are carried over to the
 * InvoiceItem.
 */
changer.execute()
```

## Removing an Entry Object

When modifying an existing charge with `ChargeInstallmentChanger`, the `Entry` objects are associated with existing invoice items. The `Entry` class supports a `remove` method which will ultimately remove its associated invoice item. However, removing certain invoice items can cause data corruption. An invoice item cannot be removed if any of the following conditions exists.

- The invoice item has been billed.
- The charge associated with the invoice item has been reversed.
- Commission has been earned or written off for the invoice item.

These conditions are evaluated in the `Entry` class `canRemove` method. If any of the conditions exists, the method returns `false`. Before removing an `Entry` object, configuration code must first call `canRemove` to verify that the object and its associated invoice item can be safely removed. To prevent data corruption, the `remove` method itself verifies that the associated invoice item can be removed and throws an `IllegalStateException` if the item cannot be removed.

An `Entry` object that cannot be removed can still be reversed provided it has no pending modifications. The recommended code pattern is shown below.

```
ChargeInstallmentChanger.Entry    anEntry; // Sample ChargeInstallmentChanger.Entry object to remove  
if (anEntry.canRemove()) {  
    anEntry.remove();  
} else {  
    if( anEntry.canReverse()) {  
        anEntry.reverse();  
    }  
}
```

## Extension Properties for Charge and InvoiceItem Objects

The `Charge` and `InvoiceItem` classes, like other BillingCenter classes, can be extended by defining new properties. These extension properties present a unique situation that requires special handling during the charge invoicing process.

The `ChargeInitializer` class includes a `charge` property of type `Charge`. The `ChargeInitializer.Entry` and `ChargeInstallmentChanger.Entry` objects are ultimately converted to `InvoiceItem` objects near the completion of the charge invoicing process. Object instances of type `Charge` or `InvoiceItem` do not exist during much of the invoicing process. This presents a question about how to manipulate the values of extension properties for objects that do not yet exist.

The methods described in this section provide support for extension properties of a future `Charge` or `InvoiceItem` object that does not exist. When the object is ultimately created, the property values assigned to it before its creation, including the extension property values, are carried over to the new object.

The methods can be called by a `ChargeInitializer` object to manage extension property values for a future `Charge` object. Similarly, the methods enable a `ChargeInitializer.Entry` or `ChargeInstallmentChanger.Entry` object to manage extension property values for a future `InvoiceItem` object.

The `ChargeInitializer` and `ChargeInitializer.Entry` objects support the following methods to manipulate extension properties for `Charge` and `InvoiceItem` objects, respectively.

- `setExtensionProperty`
- `getExtensionProperty`
- `addToExtensionArrayProperty`
- `removeFromExtensionArrayProperty`
- `getExtensionArrayProperties`

The `ChargeInstallmentChanger.Entry` object supports the following methods to manipulate extension properties for `InvoiceItem` objects.

- `setExtensionProperty`
- `getExtensionProperty`
- `addToExtensionArrayProperty`

- `getAddedExtensionArrayProperties`
- `removeFromExtensionArrayProperty`
- `getRemovedExtensionArrayProperties`

The remainder of this section describes the methods that support extension properties. For additional information, refer to the *Gosu API Reference*.

## Non-Array Type Extension Properties

Extension properties of types other than Array are handled by the methods `setExtensionProperty` and `getExtensionProperty`.

```
void      setExtensionProperty( objectProperty, propertyValue )
propertyValue getExtensionProperty( objectProperty )
```

For a `ChargeInitializer` object, the property is associated with its future `Charge` object which does not yet exist. For a `ChargeInitializer.Entry` or `ChargeInstallmentChanger.Entry` object, the property is associated with its future `InvoiceItem` object.

The following Gosu example defines the `MyCustomChargeType` property for a future, but currently non-existent, `Charge` object associated with a `ChargeInitializer` object.

```
/* The initializer variable is of type ChargeInitializer. The property value will be carried
 * over to the Charge object when the object is created.
 */
initializer.setExtensionProperty( Charge#MyCustomChargeType, "One-time" )
String customChargeType = initializer.getExtensionProperty( Charge#MyCustomChargeType )
```

A similar technique is employed to manage the extension properties of a future `InvoiceItem`. The only differences are that the parent object is of type `Entry` (from either a `ChargeInitializer` or `ChargeInstallmentChanger` object) and the relevant property is contained in a future `InvoiceItem` object.

The following Gosu example defines the `MyCustomItemType` property for an `InvoiceItem` object associated with an `Entry` object. The `Entry` object can be associated with either a `ChargeInitializer` or `ChargeInstallmentChanger` object.

```
/* The entries[] array contains Entry objects and can be associated with either a ChargeInitializer
 * or ChargeInstallmentChanger object. The property value will be carried over to the InvoiceItem
 * when the Entry is converted.
 */
entries[0].setExtensionProperty( InvoiceItem#MyCustomItemType, "Special fee" )
String customItemType = entries[0].getExtensionProperty( InvoiceItem#MyCustomItemType )
```

## Array Type Extension Properties

Extension properties of type `Array` are processed by a variety of methods. Some of the methods are available for all three of the relevant objects—`ChargeInitializer`, `ChargeInitializer.Entry`, and `ChargeInstallmentChanger.Entry`—and some methods are available for only one or two of the objects.

### Adding To and Removing From an Array Extension Property

A `Charge` or `InvoiceItem` object with extension properties of type `Array` can add and remove array elements by calling the methods `addToExtensionArrayProperty` and `removeFromExtensionArrayProperty`.

```
void addToExtensionArrayProperty( objectProperty[], propertyValue )
void removeFromExtensionArrayProperty( objectProperty[], propertyValue )
```

For a `ChargeInitializer` object, the array property is associated with its future `Charge` object which does not yet exist. For a `ChargeInitializer.Entry` or `ChargeInstallmentChanger.Entry` object, the array property is associated with its `InvoiceItem` object. In all cases, the `objectProperty[]` parameter references an array extension property on the associated `Charge` or `InvoiceItem` object. The `propertyValue` parameter is added to or removed from the specified `objectProperty[]` array.

`ChargeInstallmentChanger.Entry` objects are handled in a special manner. A `ChargeInstallmentChanger` is used to modify an existing charge, so the `Entry` object may be related to an existing `InvoiceItem` that contains populated array extension properties. Rather than load every element for every `InvoiceItem` array extension property, the `ChargeInstallmentChanger.Entry` maintains two groups—one for elements added to array extension properties and another for removed elements. When the charge is ultimately executed, the groups of added and removed array elements are applied to the affected `InvoiceItem` array extension properties.

Special handling is provided to prevent array elements from existing in both groups. If an element is added that had earlier been removed, the following steps are performed.

- The element is taken out of the removed-elements group
- The element is placed in the added-elements group

A similar operation is performed on removed array elements that had been added earlier.

- The element is taken out of the added-element group
- The element is placed in the removed-element group

The array elements contained in a group can be retrieved by the methods `getAddedExtensionArrayProperties` and `getRemovedExtensionArrayProperties`. For additional information, see the section “Retrieving Elements Added To or Removed From An Array Extension Property” on page 432.

The following Gosu example adds an element to the `MyCustomOverrides` array extension property for a future Charge of a `ChargeInitializer` object. The second code statement removes the element from the array.

```
/* The initializer variable is of type ChargeInitializer. The overrideObject is of any desired type.  
 * The element added to the array is carried over to the Charge object when the Charge object is  
 * created.  
 */  
initializer.addToExtensionArrayProperty( Charge#MyCustomOverrides, overrideObject )  
initializer.removeFromExtensionArrayProperty( Charge#MyCustomOverrides, overrideObject )
```

A similar technique is employed to manage the array extension properties of an `InvoiceItem`. The only differences are that the parent object is of type `Entry` (from either a `ChargeInitializer` or `ChargeInstallmentChanger` object) and the relevant property is contained in an `InvoiceItem` object.

The following Gosu example adds an `ItemEvent` object to the `MyCustomEvents` array extension property for an `InvoiceItem` object associated with an `Entry` object. The `Entry` object can be associated with either a `ChargeInitializer` or `ChargeInstallmentChanger` object. The second code statement removes the object from the array.

```
/* The entries[] array contains Entry objects and can be associated with either a  
 * ChargeInitializer or ChargeInstallmentChanger object.  
 */  
entries[0].addToExtensionArrayProperty( InvoiceItem#MyCustomEvents, itemEventObject )  
entries[0].removeFromExtensionArrayProperty( InvoiceItem#MyCustomEvents, itemEventObject )
```

## Retrieving an Array Extension Property

A Charge or `InvoiceItem` object with extension properties of type `Array` can retrieve the array elements by calling the method `getExtensionArrayProperties`.

```
List<Object> getExtensionArrayProperties( objectProperty[] )
```

The method returns the list of elements currently stored in the specified `objectProperty[]` array of the Charge or `InvoiceItem` object.

For a `ChargeInitializer` object, the array extension property is associated with its future Charge object which does not yet exist. For a `ChargeInitializer.Entry` object, the array property is associated with its future `InvoiceItem` object. The `ChargeInstallmentChanger.Entry` class does not support this method.

The following Gosu example retrieves the list of elements stored in the `MyCustomOverrides` array extension property for a future Charge of a `ChargeInitializer` object.

```
/* The initializer variable is of type ChargeInitializer */  
listOfOverrides = initializer.getExtensionArrayProperties( Charge#MyCustomOverrides )
```

A similar technique is employed to retrieve the elements of an array extension property of a future `InvoiceItem`. The only differences are that the parent object is of type `ChargeInitializer.Entry` and the relevant extension property is contained in a future `InvoiceItem` object.

The following example retrieves the list of `ItemEvent` objects stored in the `MyCustomEvents` array extension property for a future `InvoiceItem` of a `ChargeInitializer.Entry` object.

```
/* The entries[] array contains Entry objects associated with a ChargeInitializer object */
List<ItemEvent> listOfItemEvents = entries[0].getExtensionArrayProperties( InvoiceItem#MyCustomEvents )
```

### Retrieving Elements Added To or Removed From An Array Extension Property

A `ChargeInstallmentChanger.Entry` object with an extension property of type `Array` can retrieve the elements added to or removed from the array by calling the methods `getAddedExtensionArrayProperties` and `getRemovedExtensionArrayProperties`.

```
List<Object> getAddedExtensionArrayProperties( ObjectProperty[] )
List<Object> getRemovedExtensionArrayProperties( ObjectProperty[] )
```

The `ObjectProperty[]` parameter references an array extension property of an `InvoiceItem` associated with a `ChargeInstallmentChanger.Entry` object. The `ChargeInitializer` and `ChargeInitializer.Entry` classes do not support these methods.

Each method returns the list of elements that have been added to or removed from the specified array extension property. For additional details, refer to the section “Adding To and Removing From an Array Extension Property” on page 430.

The following Gosu example retrieves the list of `ItemEvent` objects that were added to the `MyCustomEvents` array extension property of an `InvoiceItem` object. The second code statement retrieves the list of removed `ItemEvent` objects from the same array property.

```
/* The entries[] array contains Entry objects associated with a ChargeInstallmentChanger object */
List<ItemEvent> listOfAddedEvents = entries[0].getAddedExtensionArrayProperties( InvoiceItem#MyCustomEvents )
List<ItemEvent> listOfRemovedEvents = entries[0].getRemovedExtensionArrayProperties( InvoiceItem#MyCustomEvents )
```

## Charge Invoicing and Payment Plans

The charge invoicing process is closely related to the active payment plan. When creating a new charge, the payment plan is used to determine invoice dates, the frequency of invoices, and many other results of the charge invoicing process.

Similarly, when the payment plan is altered, the charge invoicing process is used to recalculate the invoices and items on the invoices. Because a change to the payment plan affects existing charges, the recalculation of invoices and invoice items is very similar to modifying an existing, executed charge. For details, refer to “Modifying an Existing Charge” on page 427.

Specifically, a change to the payment plan uses the `ChargeInstallmentChanger` class to enable modifications to the existing invoice items of a particular charge. During the execution of the payment plan recalculation, the `ChargeInstallmentChanger` object is managed by an instance of the `PaymentPlanChanger` class. The following code sample demonstrates the use and relationship of the `PaymentPlanChanger` and `ChargeInstallmentChanger` objects. For further details about these classes, refer to the *Gosu API Reference*.

```
/* Create the PaymentPlanChanger object */
PaymentPlanChanger ppChanger = new PaymentPlanChanger( policyPeriod,
   newPaymentPlan,
   paymentHandling,
   itemsToReverseOrRemove )

/* Retrieve the list of Entry objects in the recalculated payment plan */
List<ChargeInstallmentChanger.Entry> allEntries = ppChanger.getInstallmentPreview()

/* Iterate through the Entry objects, modifying them as desired */
for( ChargeInstallmentChanger.Entry entry : allEntries ){
    /* Modify Entry object. In this case, we simply calculate the sum of all Entry objects. */
    totalAmount = totalAmount.add( entry.getAmount() )
}
```

## Payment Plan Modifiers

A billing instruction can alter the payment plan behavior by including a payment plan modifier. For details, see “Payment Plan Modifiers” on page 119 in the *Application Guide*.

In practice, payment plan modifiers are rarely necessary. The preferred method for altering payment plan behavior is to customize the Charge Invoicing Process. For special cases where customizing the Charge Invoicing Process is not possible, a payment plan modifier can be used.

### Processing Payment Plan Modifiers in BillingCenter

To process a custom payment plan modifier in BillingCenter:

1. Implement the payment plan modifier in a Gosu class named <ppm>MethodsImpl, where <ppm> represents the modifier’s name. The implementation class must extend the `PaymentPlanModifierMethodsImpl` class, which implements the `PaymentPlanModifierMethods` interface.

You can use the predefined modifiers’ implementation classes as models. See the Gosu files in `<BillingCenter-directory>/modules/configuration/gsrc/gw/paymentplanmodifier`.

2. Create a metadata entity named <ppm>.eti. This entity defines the modifier and identifies the implementation class.

You can use the predefined metadata entity files as models. See, for example, `config/Metadata/Entity/DownPaymentOverride.eti`.

When BillingCenter encounters a billing instruction that contains an instance of the new payment plan modifier, it constructs an instance of the implementation class and calls its `modify` method. This method receives a copy of the payment plan as a parameter, and may modify it in any way necessary to accomplish the payment plan modifier’s purpose.

### Accessing Payment Plan Modifiers

In a billing instruction entity, the `PaymentPlanModifiers` field contains an array key to `PaymentPlanModifier`.

A billing instruction entity is a concrete entity descended from `P1cyBillingInstruction`, such as `Issuance` or `Renewal`.

The payment plan modifier types and their entity names are described in the following table.

Payment Plan Modifier Type (also the Entity Name)	Description
ChargeSlicingModifier	Overrides any of the payment plan’s charge slicing settings. It uses a <code>ChargeSlicingOverrides</code> , which specifies which settings to override.
DownPaymentOverride	Overrides the <code>Down Payment %</code> field of the payment plan by setting <code>PaymentPlan.DownPaymentPercent</code> .
MatchPlannedInstallments	Matches the planned installments related to the charge. <ul style="list-style-type: none"><li>• Sets <code>PaymentPlan.MaximumNumberOfInstallments</code> to the number of planned installments for the issuance charge</li><li>• Sets <code>PaymentPlan.DownPaymentPercent</code> to 0</li><li>• Sets other miscellaneous <code>PaymentPlan</code> fields</li></ul>
MaximumNumberOfInstallmentsOverride	Overrides the payment plan’s <code>Max # Installments</code> field by setting <code>PaymentPlan.MaximumNumberOfInstallments</code> .
SuppressDownPayment	Suppresses any down payment by setting <code>PaymentPlan.DownPaymentPercent</code> to 0 (zero).

## InvoiceStream and DateSequence Plugins

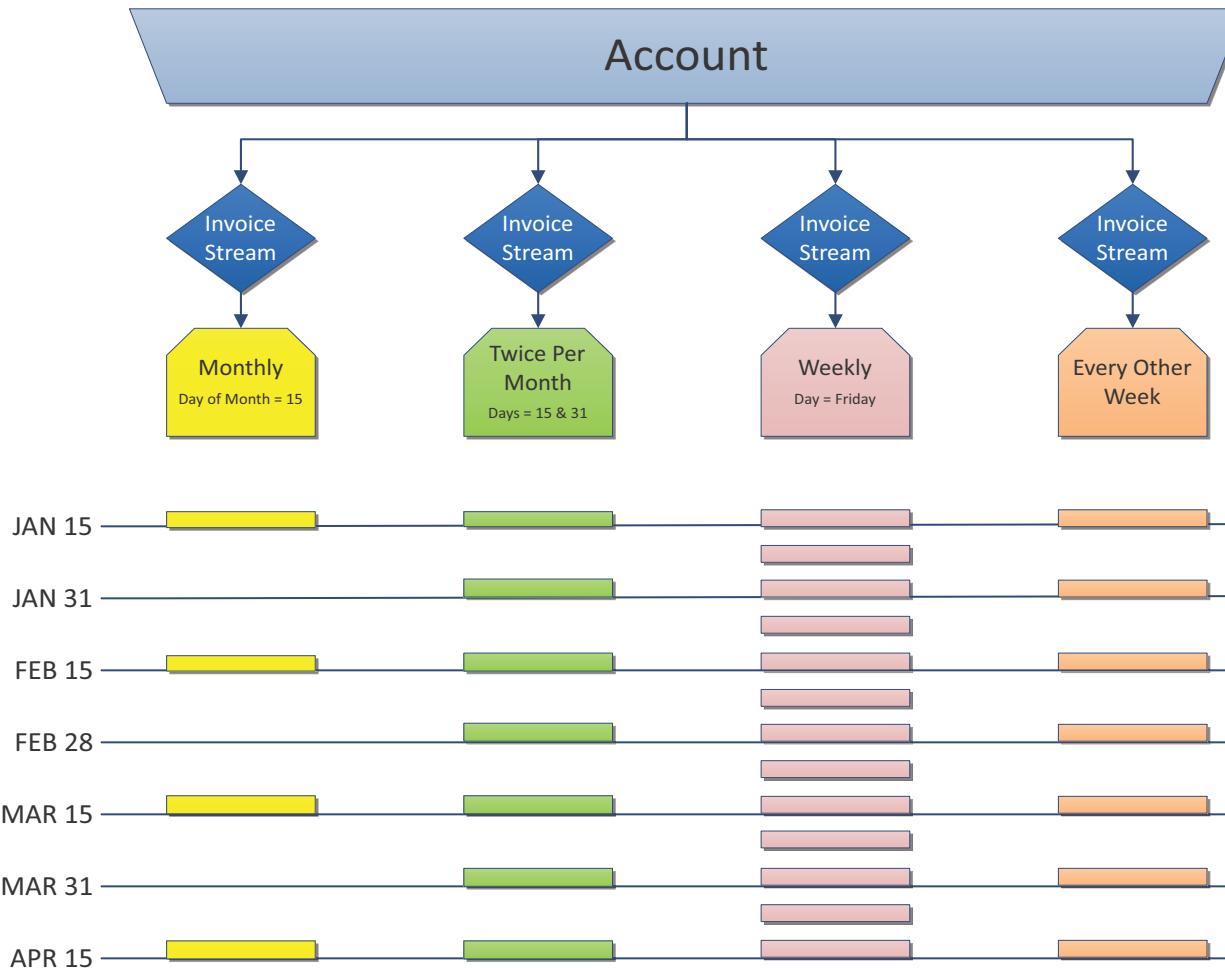
The *invoice stream* comprises all invoices for a single policy period. Invoice streams never cross over to other policy periods. Invoices contain individual invoice items. Each item can be placed on or moved to any invoice within the invoice stream; they cannot be placed on or moved to invoices in other streams.

An invoice stream is organized by a *date sequence*. A date sequence is a series of dates based on a time interval or *periodicity*, such as a week or a month. BillingCenter examples of periodicities include **Monthly**, **Weekly**, and **Twice Per Month**.

The *anchor date* is related to a date sequence. The anchor date is the day of the month used as the basis for generating a date sequence, such as the first day of the month. For example, if the anchor date is the 15th, a **Monthly** date sequence produces invoices on the 15th of every month—January 15, February 15, and so on.

The date sequences **Monthly**, **Every Other Month**, **Quarterly**, **Every Four Months**, and **Every Six Months** use one anchor date. The **Twice Per Month** periodicity requires two anchor dates—one for each invoice date.

The following diagram demonstrates the relationship between invoice streams, date sequences, and anchor dates. The invoice streams depicted use four common date sequences: Monthly, Twice Per Month, Weekly, and Every Other Week. Although each invoice stream can have a different anchor date, the Monthly and Twice Per Month streams shown in the diagram use the same anchor date—the 15th of the month. The Twice Per Month stream includes a second anchor date of the 31st of the month. The Weekly stream uses an anchor date of Friday.



BillingCenter provides two plugins that enable you to configure invoice streams and date sequences. The following table provides a brief description of each plugin.

Plugin	Description
InvoiceStream.gs	Called when the <code>InvoiceAssembler</code> entity creates an invoice. The Invoice Stream plugin selects the invoice stream to use for the invoice. In the base configuration, the plugin returns the appropriate default invoice stream based on the periodicity. See “Invoice Stream Plugin” on page 180 in the <i>Integration Guide</i> .
DateSequence.gs	Called by the Invoice Stream plugin. The Date Sequence plugin creates a date sequence that contains the anchor date and specifies the date interval or <i>periodicity</i> . The plugin can be customized to return a date sequence other than the default. It can also be extended to recognize custom periodicities. See “Date Sequence Plugin” on page 183 in the <i>Integration Guide</i> .





## chapter 35

# Configuring Payment Allocation Plans

This topic describes how to configure BillingCenter payment allocation plans.

This topic includes:

- “Payment Allocation Plan Configuration Overview” on page 437
- “Writing an Invoice Item Filter” on page 437
- “Ordering of Invoice Items” on page 439
- “Modifying the DirectBillPayment Plugin” on page 441

**See also**

- “Payment Allocation Plans” on page 123 in the *Application Guide*

## Payment Allocation Plan Configuration Overview

Payment allocation plans can be created and modified from the BillingCenter user interface, as described in “Payment Allocation Plans” on page 123 in the *Application Guide*. If you desire a special invoice item filter or a unique ordering of invoice items not available in the base configuration, it can be written using the Gosu programming language.

## Writing an Invoice Item Filter

An invoice item filter determines whether a particular invoice item is eligible to be paid. This section describes the programming tasks necessary to create a special invoice item filter. A sample filter written in Gosu is also provided.

### Programming Tasks

The following tasks outline the steps necessary to write a special invoice item filter for a payment allocation plan:

1. Define a new typecode in the `DistributionFilterType` typelist.
2. Create a thread-safe class that implements the `DistributionFilterCriterion` interface.

3. Implement the new class. This entails writing (a) the TypeKey property to return the newly-defined typecode and (b) the restrict method to perform the desired filter operation.
4. Register the class in the `LinkedImplementationLoaderImpl.returnDistributionFilterCriteria` method.

## Sample Invoice Item Filter in Gosu

This section describes the implementation of a sample invoice item filter using the Gosu programming language. The filter accepts invoice items associated with a delinquent policy period and rejects items in a non-delinquent policy period.

### Define the Filter Typecode

A new typecode for the filter must be added to the `DistributionFilterType` typelist.

1. If the `DistributionFilterType` typelist has not already been extended, right-click `configuration` → `config` → `Metadata` → `Typelist` → `DistributionFilterType.tii` and select `New` → `Typelist Extension`.
2. If the `DistributionFilterType` typelist has already been extended, select `configuration` → `config` → `Extensions` → `Typelist` → `DistributionFilterType.ttx`.
3. Add a new typecode to the extended typelist. For the sample filter, set the typecode fields as shown below.
  - `code`: `DelinquentPolicyPeriod`
  - `name`: Delinquent Policy Period
  - `desc`: Accept invoice items if policy period is delinquent

### Define the Filter Class

A new class for the filter must be defined. The class must implement the `DistributionFilterCriterion` interface, and the class implementation must be thread-safe.

1. A recommended best practice places configuration code in your own packages. The sample filter is written for an imaginary company called Acme. Accordingly, its payment allocation configuration code is placed in a package named `acme.payment`. To create the package, right-click `configuration` → `gsrc` and select `New` → `Package`. Name the package `acme.payment`.
2. Define the new filter class. Right-click `configuration` → `gsrc` → `acme.payment` and select `New` → `Gosu class`. Name the class `DelinquentPolicyPeriodFilterCriterion`.
3. The new filter class must implement the `DistributionFilterCriterion` interface. The empty skeleton definition for the filter class is shown below.

```
package acme.payment
uses gw.bc.payment.DistributionFilterCriterion
class DelinquentPolicyPeriodFilterCriterion implements DistributionFilterCriterion { }
```

### Implement the TypeKey Method

The filter class must implement a getter method for the `TypeKey` property. The method must return the filter's typecode added to the `DistributionFilterType` typelist in an earlier step.

The `TypeKey` method for the sample filter is shown below.

```
override property get
TypeKey() : DistributionFilterType {
    return TC_DELINQUENTPOLICYPERIOD
}
```

## Implement the restrict Method

The filter class must implement the `restrict` method which performs the actual filtering operation and returns the group of invoice items that passed through the filter. The returned invoice items are eligible to be paid. The `restrict` method must be thread-safe.

The `restrict` method for the sample filter uses two Gosu convenience classes provided in the base configuration: `Paths` and `RestrictionBuilder`. For information about `Paths` and `RestrictionBuilder`, see “Paths and Restriction Builders” on page 183 in the *Gosu Reference Guide*.

The sample filter implementation of the `restrict` method is shown below.

```
uses gw.api.path.Paths
uses gw.api.restriction.RestrictionBuilder

override function
restrict( restrictions : RestrictionBuilder <InvoiceItem>,
          moneyRcvd    : DirectBillMoneyRcvd ){

    // If payment is associated with a policy period AND the policy period is delinquent...
    if( moneyRcvd.PolicyPeriod != null
        &&
        moneyRcvd.PolicyPeriod.hasActiveDelinquencyProcess() ){
        // ...then accept and return all the invoice items associated with this policy period.
        restrictions.compare( Paths.make( InvoiceItem#PolicyPeriod ),
                               Equals,
                               moneyRcvd.PolicyPeriod )
    }
}
```

## Register the Filter Class

The filter class must be registered in the `LinkedImplementationLoaderImpl.returnDistributionFilterCriteria` method. This entails adding a code statement to the `returnDistributionFilterCriteria` method. The code statement must construct a new instance of the filter class. The following steps register the sample filter class.

1. Select **Navigate → Class...** and enter `LinkedImplementationLoaderImpl` to load the class's implementation file.
2. Register the sample filter class in the `returnDistributionFilterCriteria` method. The code is shown below.

```
uses acme.payment.DelinquentPolicyPeriodFilterCriterion // Sample filter class

override function
returnDistributionFilterCriteria() : Collection<DistributionFilterCriterion> {
    return {
        new PositiveDistributionFilterCriterion(),
        // ... other filter registrations ...
        new PastDueDistributionFilterCriterion(),
        // Register sample filter
        new DelinquentPolicyPeriodFilterCriterion()
    }
}
```

## Ordering of Invoice Items

Invoice items eligible for payment are grouped and ordered based on the priority in which they will be paid. Items with a higher priority are paid before items having a lower priority. This section describes the programming tasks necessary to create a special grouping and ordering of eligible invoice items. A sample implementation written in Gosu is also provided.

## Programming Tasks

The following tasks outline the steps necessary to write a special grouping and ordering of eligible invoice items for a payment allocation plan:

1. Define a new typecode in the `InvoiceItemOrderingType` typelist.
2. Create a thread-safe class that extends `DirectInvoiceItemAllocationOrdering`.
3. Implement the new class. This entails writing (a) the `TypeKey` property to return the newly-defined typecode and (b) the `compare` method which accepts two invoice items and returns the result of comparing them.
4. Register the class in the `LinkedImplementationLoaderImpl.returnPaymentAllocationOrderings` method.

## Sample Ordering of Invoice Items in Gosu

This section describes the implementation of a sample ordering of eligible invoice items using the Gosu programming language. The sample prioritizes items based on their charge type. Premium charges have the highest priority and are paid first. Tax charges are paid second. Fee charges have the lowest priority.

### Define the Ordering Typecode

A new typecode for the ordering must be added to the `InvoiceItemOrderingType` typelist.

1. If the `InvoiceItemOrderingType` typelist has not already been extended, right-click `configuration` → `config` → `Metadata` → `Typelist` → `InvoiceItemOrderingType.tti` and select `New` → `Typelist Extension`.
2. If the `DistributionFilterType` typelist has already been extended, select `configuration` → `config` → `Extensions` → `Typelist` → `InvoiceItemOrderingType.ttx`.
3. Define a new typecode to the extended typelist. For the sample filter, set the typecode fields as shown below.
  - `code: ChargeType`
  - `name: Charge Type`
  - `desc: Order eligible invoice items based on the Charge Type (Premium, Tax, Fee)`

### Define the Ordering Class

A new class for the ordering must be defined. The class must extend `DirectInvoiceItemAllocationOrdering`, and the class implementation must be thread-safe.

1. A recommended best practice places configuration code in your own packages. The sample ordering is written for an imaginary company called Acme. Accordingly, its payment allocation configuration code is placed in a package named `acme.payment`. To create the package, right-click `configuration` → `gsrc` and select `New` → `Package`. Name the package `acme.payment`.
2. Define the new ordering class. Right-click `configuration` → `gsrc` → `acme.payment` and select `New` → `Gosu class`. Name the class `ChargeTypeOrdering`.
3. The new ordering class must extend `DirectInvoiceItemAllocationOrdering`. The empty skeleton definition for the filter class is shown below.

```
package acme.payment  
uses gw.payment.DirectInvoiceItemAllocationOrdering  
class ChargeTypeOrdering extends DirectInvoiceItemAllocationOrdering {  
}
```

### Implement the TypeKey Method

The ordering class must implement a getter method for the `TypeKey` property. The method must return the ordering's typecode added to the `InvoiceItemOrderingType` typelist in an earlier step.

The TypeKey method for the sample ordering is shown below.

```
override property get
TypeKey() : InvoiceItemOrderingType {
    return TC_CHARGETYPE
}
```

### Implement the compare Method

The ordering class must implement the compare method which examines the charge types of two invoice items and returns a value indicating the priority relationship of the two items. If the first invoice item has a higher priority than the second item, the method returns a value greater than zero. If the first item has a lower priority than the second item, the method returns a value less than zero. If the two items have the same priority, the method returns zero. The compare method must be thread-safe.

The sample implementation of the compare method is shown below.

```
uses java.lang.Integer

override function
compare( left : InvoiceItem, right : InvoiceItem ) : int {
    var listByPriority = [
        ChargeCategory.TC_PREMIUM,
        ChargeCategory.TC_TAX,
        ChargeCategory.TC_FEE
    ]
    var leftPriority = listByPriority.indexOf( left.Charge.ChargePattern.Category )
    var rightPriority = listByPriority.indexOf( right.Charge.ChargePattern.Category )
    return Integer.compare( rightPriority, leftPriority )
}
```

### Register the Ordering Class

The ordering class must be registered in the

`LinkedImplementationLoaderImpl.returnPaymentAllocationOrderings` method. This entails adding a code statement to the `returnPaymentAllocationOrderings` method. The code statement must construct a new instance of the ordering class. The following steps register the sample ordering class.

1. Select **Navigate** → **Class...** and enter `LinkedImplementationLoaderImpl` to load the class's implementation file.
2. Register the sample ordering class in the `returnPaymentAllocationOrderings` method. The code is shown below.

```
uses acme.payment.ChargeTypeOrdering // Sample ordering class

override function
returnPaymentAllocationOrderings() : Collection<InvoiceItemAllocationOrderings> {
    return [
        new BilledDateOrdering(),
        // ... other ordering registrations ...
        new RecaptureFirstOrdering(),
        // Register sample ordering
        new ChargeTypeOrdering()
    ]
}
```

## Modifying the DirectBillPayment Plugin

The code that carries out the payment allocation plan, including the plan's invoice item filtering and item ordering, is invoked by the `DirectBillPayment` plugin. For plugin details, see "Direct Bill Payment Plugin" on page 192 in the *Integration Guide*.

It is reasonable to expect the final payment distribution to conform to the defined settings in the payment allocation plan. However, because the `DirectBillPayment` plugin hooks into certain allocation plan operations, if the plugin methods are customized, the distribution results can differ from the expected results. For example, the plugin might perform special processing before invoking the payment allocation plan. Alternatively, the plugin might invoke the allocation plan two or more times, passing the plan different invoice items in each invocation.

The `DirectBillPayment` plugin's `allocatePayment` method invokes the payment allocation plan. Other methods in the plugin also invoke the payment allocation plan, but those methods handle very special situations. The discussion in this section focuses on customizing the `allocatePayment` method.

## The `allocatePayment` Method

To view and modify the `allocatePayment` method, select **Navigate** → **Class...** and enter `DirectBillPayment`. Select the plugin entry. The `allocatePayment` implementation in the base configuration is shown below. The method first determines the total amount of money to distribute. It then passes that amount and the list of all invoice items associated with the payment to the payment allocation plan for processing. The payment allocation plan filters the eligible invoice items, orders them, and finally distributes money among them.

```
override function
allocatePayment( payment : DirectBillPayment, amount : MonetaryAmount ){
    // Determine total amount to distribute
    var amountToDistribute = AllocationPool.withGross( amount )
    // Invoke payment allocation plan
    paymentAllocationStrategy( payment ).allocate( payment.DistItemsList, amountToDistribute )
}
```

A sample modification to the `allocatePayment` method is shown below. This version filters worker's comp invoice items and pays those items by invoking the payment allocation plan. If any money remains after paying the worker's comp items, the remaining invoice items are collected and passed to a second invocation of the allocation plan. Comments in the source code describe the operations performed.

```
override function
allocatePayment( payment : DirectBillPayment, amount : MonetaryAmount ){

    // Collect all worker's comp invoice items
    var workersCompItems = payment.DistItems.
        where(\ item -> item.PolicyPeriod.Policy.LOBCode == "WorkersComp" )

    // If any worker's comp invoice items exist...
    if( !workersCompItems.IsEmpty ){
        // ...pay them first by invoking the payment allocation plan
        // Note: The plan's filter may still determine these items to be ineligible for payment
        var amountToDistribute = AllocationPool.withGross( amount )
        paymentAllocationStrategy( payment ).allocate( workersCompItems.toList(),
            amountToDistribute )
    }

    // Collect all remaining invoice items
    var otherItems = payment.DistItems.
        where(\ item -> item.PolicyPeriod.Policy.LOBCode != "WorkersComp" )

    // If any remaining invoice items exist...
    if( !otherItems.IsEmpty ){

        // If any worker's comp items existed...
        var workersCompAmount = new MonetaryAmount( 0, amount.Currency ) // default amount distributed
        if( !workersCompItems.IsEmpty ){
            // ...calculate how much was distributed to them
            workersCompAmount = workersCompItems.sum(\ item -> item.GrossAmountOwed )
        }

        // Calculate the remaining money to distribute
        var remainingMoney = amount - workersCompAmount
        var zero = Obd.ofCurrency( amount.Currency ) // constant in current currency
        var amountToDistribute = AllocationPool.withGross( remainingMoney > zero ? remainingMoney : zero )

        // ...pay the remaining invoice items using any remaining money
        paymentAllocationStrategy( payment ).allocate( otherItems.toList(), amountToDistribute )
    }
}
```



## chapter 36

# Configuring Distributions

This topic describes how to configure BillingCenter payment and credit distributions.

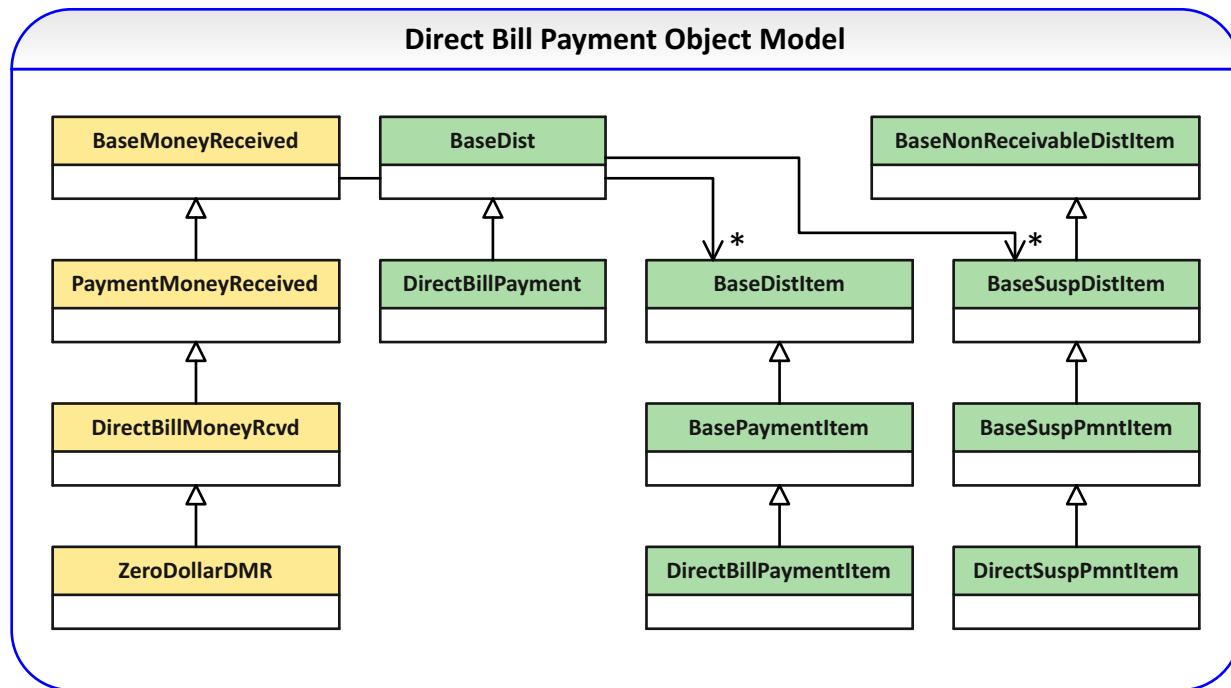
This topic includes:

- “Modifying Direct Bill Payment and Credit Distributions” on page 444
- “Modifying Agency Bill Payment and Credit Distributions” on page 445
- “Modifying Agency Bill Promise Distributions” on page 446

# Modifying Direct Bill Payment and Credit Distributions

## Direct Bill Payment Object Model

BillingCenter has a unified object model for direct bill and agency bill payments and payment distribution. The following graphic focuses on direct bill payments. The object model for agency bill payments follows a similar structure.



Payments made to the carrier are tracked in BillingCenter as *direct bill money received*. The insured remits the gross amount to the carrier. This is the amount used to make a payment. The carrier then makes commission payments to any producers associated with the account.

The **DirectBillMoneyRcvd** entity holds details about the payment, including the account that made the payment. The **DirectBillPayment** entity tracks the amount of money that has been distributed (if any) and the amount available to distribute. It has an array of distribution items represented as **DirectBillPaymentItem** entities. Each **DirectBillPaymentItem** contains details of distribution amounts for the invoice item and points to the invoice item that is being paid by this item. Distribution amounts include gross amounts owed, gross amounts to apply, commission amount owed, and commission amount to apply.

In summary, for each payment there is a payment item that targets an invoice item and indicates how much gross and commission money to apply to the invoice item.

## Example Modification of a Direct Bill Payment and Credit Distribution

Payment and credit distributions can be modified using the Gosu programming language or the BillingCenter user interface. For information about utilizing the user interface for payment distributions, see “[Modifying Direct Bill Payments](#)” on page 298 in the *Application Guide*. To modify a credit distribution with the user interface, see “[Creating and Editing Direct Bill Credit Distributions](#)” on page 295 in the *Application Guide*.

When modifying a direct bill payment or credit distribution in Gosu, the elements of the executed distribution must not be altered directly. Instead, a copy of the distribution must be created. All modifications are then performed upon the copy. After completing the desired modifications, the copy can be executed and committed. The newly-committed distribution becomes the payment or credit of record, replacing the original distribution. The original distribution is accordingly reversed.

The sample code below demonstrates the operations performed to modify a direct bill payment or credit distribution. Notice that the identical procedure is performed whether the distribution is a payment or a credit. The classes and methods used in the sample are described in the *Gosu API Reference*.

```
/* Variables */
DirectBillPayment    executedDist      // An executed payment or credit distribution
BaseDistItem          distItem        // An invoice item in executedDist
DirectBillPayment    modifiedDist     // A copy of executedDist

/* Here is what NOT to do: Do NOT directly modify fields in the executed distribution. */
distItem = executedDist.getDistItemFor( /* Some invoice item */ )
distItem.setGrossAmountToApply( $100 ) // Do NOT do this!!!

/* Instead, do this: Create a copy of the executed distribution and modify the copy */
modifiedDist = DirectBillPaymentFactory.createModifyingDirectPayment( executedDist )

/* Grab an invoice item you want to modify from the copied distribution */
distItem = modifiedDist.getDistItemFor( /* Some invoice item */ )

/* Change the amount allocated to this invoice item */
distItem.setGrossAmountToApply( $100 )

/* Execute and commit the modified distribution, making it the payment or credit of record */
modifiedDist.execute()
modifiedDist.getBundle().commit()
```

## Modifying Agency Bill Payment and Credit Distributions

Agency bill payment and credit distributions can be modified using the Gosu programming language or the BillingCenter user interface. For information about utilizing the user interface for payment distributions, see “Agency Bill Payments Handling” on page 348 in the *Application Guide*. To modify a credit distribution with the user interface, see “Agency Bill Credit Distributions Handling” on page 352 in the *Application Guide*.

When modifying an agency bill payment or credit distribution in Gosu, each distribution comprises three elements: a *money* element, a *distribution* element, and one or more *distribution items*. The *money* element wraps the total monetary amount of the distribution. The *distribution* element contains details about the transaction, including an array of distribution items. Each distribution item refers to a particular invoice item and specifies the monetary amount distributed to the invoice item. When editing a particular distribution, each of the three elements can be modified.

The three elements of the distribution being modified must not be altered directly. Instead, a copy of each element must be created. All modifications are then performed upon the copies. After completing the desired modifications, the copies can be executed and committed. The newly-committed distribution becomes the payment or credit of record, replacing the original distribution. The original distribution is accordingly reversed.

The sample code below demonstrates the operations performed to modify an agency bill payment or credit distribution. Notice that the identical procedure is performed whether the distribution is a payment or a credit. The classes and methods used in the sample are described in the *Gosu API Reference*.

```
/* Variables */
AgencyBillMoneyRcvd          distMoney      // Executed distribution to modify
AgencyBillPaymentMoneySetup   moneySetup    // Wrapper for executed distribution. Includes link to
  // original distribution element and a copy of money
  // element.
  // Copy of money element. Retrieved from moneySetup.
AgencyBillMoneyRcvd          copyMoney     // Copy of distribution element. Created from moneySetup.
AgencyCycleDist               copyDist      // Copy of distributed item. Created from moneySetup.

/* Retrieve a wrapper of the executed distribution.
 * This wrapper includes the original distribution element and a copy of the money element.
 */
*/
```

```

moneySetup = AgencyBillMoneySetupFactory.createEditingPaymentMoney( distMoney,
    distMoney.getBundle() )

/* If you want to edit the money element, retrieve the copy of it from the moneySetup wrapper */
copyMoney = (AgencyBillMoneyRcvd)moneySetup.getMoney()          // Get the copy
copyMoney.setAmount( $120 )                                     // Edit the copy

/* Make a copy of the linked distribution element, including its distribution items */
copyDist = moneySetup.prepareDistribution()

/* Retrieve the copies of the distribution items and edit the copies */
for( copyDistItem : copyDist.getDistItems() ){                  // Get a copy
    copyDistItem.setGrossAmountToApply( $10 )                   // Edit the copy
}

/* Execute and commit the modified copies. The modified distribution becomes the distribution of
 * record, replacing the original distribution.
 */
copyDist.execute()
copyMoney.getBundle().commit()

```

## Modifying Agency Bill Promise Distributions

Agency bill promise distributions can be modified using the Gosu programming language or the BillingCenter user interface. For information about utilizing the user interface for promise distributions, see “Agency Bill Promise Handling” on page 351 in the *Application Guide*.

Modifying an agency bill promise distribution is nearly identical to modifying an agency bill payment or credit distribution. Refer to “Modifying Agency Bill Payment and Credit Distributions” on page 445 for complete information. The only differences required when modifying a promise distribution are listed below.

- The class type of the executed distribution is `PromisedMoney`.
- The class type of the wrapper for the executed distribution is `AgencyBillPromisedMoneySetup`.
- The method called to retrieve the wrapper is `createEditingPromisedMoney`.

The class types and method invocation for promise distributions are shown below.

```

/* Variables */
PromisedMoney           distMoney      // Executed distribution to modify
AgencyBillPromisedMoneySetup   moneySetup   // Wrapper for executed distribution. Includes link to
  // original distribution element and a copy of money
  // element.

/* Retrieve a wrapper of the execute distribution.
 * This wrapper includes the original distribution element and a copy of the money element.
 */
moneySetup = AgencyBillMoneySetupFactory.createEditingPromisedMoney( distMoney,
    distMoney.location.getBundle() )

```

The following code sample provides a complete example of modifying an agency bill promise distribution. The classes and methods used in the sample are described in the *Gosu API Reference*.

```

/* Variables */
PromisedMoney           distMoney      // Executed distribution to modify
AgencyBillPromisedMoneySetup   moneySetup   // Wrapper for executed distribution. Includes link to
  // original distribution element and a copy of money
  // element.
AgencyBillMoneyRcvd        copyMoney     // Copy of money element. Retrieved from moneySetup.
AgencyCyclelist            copyDist      // Copy of distribution element. Created from moneySetup.
BaseDistItem               copyDistItem   // Copy of distributed item. Created from moneySetup.

/* Retrieve a wrapper of the executed distribution.
 * This wrapper includes the original distribution element and a copy of the money element.
 */
moneySetup = AgencyBillMoneySetupFactory.createEditingPromisedMoney( distMoney,
    distMoney.location.getBundle() )

/* If you want to edit the money element, retrieve the copy of it from the moneySetup wrapper */
copyMoney = (AgencyBillMoneyRcvd)moneySetup.getMoney()          // Get the copy
copyMoney.setAmount( $120 )                                     // Edit the copy

/* Make a copy of the linked distribution element, including its distribution items */
copyDist = moneySetup.prepareDistribution()

```

```
/* Retrieve the copies of the distribution items and edit the copies */
for( copyDistItem : copyDist.getDistItems() ){
    copyDistItem.setGrossAmountToApply( $10 )           // Get a copy
}   // Edit the copy

/* Execute and commit the modified copies. The modified distribution becomes the distribution of
 * record, replacing the original distribution.
 */
copyDist.execute()
copyMoney.getBundle.commit()
```





## chapter 37

# Configuring Write-Offs

This topic describes how to configure BillingCenter write-offs.

This topic includes:

- “Commission Write-Offs” on page 449
- “Producer Write-Offs” on page 450

## Commission Write-Offs

When a portion of an account or policy charge is written off, the associated commission can be handled in various ways. For example, a proportionate amount of commission can be written off. Alternatively, the commission for the write-off amount can be earned immediately. Other methods of handling the commission are also supported. The manner in which to handle a commission for a write-off charge is defined by configuring the `Commission` plugin.

The `Commission` plugin source code is located in `configuration → gsrc → gw → plugin → commission → impl → Commission.g`. The `Commission` class defines a function called `shouldWriteOffActiveCommissionForChargeWrittenOff`. This function is called every time a charge is written off.

The boolean return value specifies how to handle the commission associated with the write-off. The interpretation of the return value also depends on the commission’s `Payable Criteria` defined in the Commission Plan. The following table describes the various combinations of return value and `Payable Criteria` and how each combination handles the commission for the write-off charge.

Return Value	Payable Criteria: Commission Handling
<code>true</code>	<code>On Payment Received</code> : Write off a commission amount proportionate to the write-off charge. <code>All other Payable Criteria settings</code> : Write off a commission amount proportionate to the write-off charge. If sufficient reserve does not exist, deduct from existing earned and paid commission.
<code>false</code>	<code>On Payment Received</code> : Earn the commission on the write-off amount. In this scenario, the write-off (as it relates to commission) is processed as though it is a payment. <code>All other Payable Criteria settings</code> : Do not affect the commission.

In the base configuration, the function always returns `false`. The function can be modified to always return `true`, or the return value can be determined based on various conditions. As an example, the following sample code returns `true` only if a commission has been paid for a written-off policy period charge.

```
override function
shouldWriteOffActiveCommissionForChargeWrittenOff( chargeWrittenOff : ChargeWrittenOff ) : boolean {

    /* Retrieve the T-Account owner of the charge */
    var tAccountOwner = chargeWrittenOff.Writeoff.TAccountOwner

    /* Is the charge owner a policy period? */
    if( tAccountOwner typeis PolicyPeriod ){
        /* Yes, any commission paid? */
        var commissionPaid = TAccountOwner.CommissionableCharges.firstWhere(
            commissionCharge->commissionCharge.PolicyCommission.CommissionSubPlan.PayableCriteria.
            equals( PayableCriteria.TC_PAYMENTRECEIVED ) )
        if( commissionPaid != null ){
            /* Yes, a commission was paid; write off part of it */
            return true
        }
    }

    /* Else... */
    return false
}
```

## Producer Write-Offs

The producer **Write-Offs** screen lists a producer's write-offs. For details, see "Producer Write-offs" on page 239 in the *Application Guide*.

### Defining a New Creation Date Filter

The write-offs listed in the producer **Write-offs** screen can be filtered based on the write-off's creation date. The base configuration provides age options for 30, 60, and other numbers of days. Additional age options can be created by customizing the `ProducerWriteOffAgeRestriction` class.

To create an age option, define a new static variable in the `ProducerWriteOffAgeRestriction` class. A portion of the class's implementation is shown below. The new variable `LAST_7_DAYS` defines a seven-day option.

```
class ProducerWriteOffAgeRestriction {

    public static var LAST_7_DAYS : ProducerWriteOffAgeRestriction =
        new ProducerWriteOffAgeRestriction(7)
    public static var LAST_30_DAYS : ProducerWriteOffAgeRestriction =
        new ProducerWriteOffAgeRestriction(30)
```

After defining the variable, add the variable to the static `ALL_OPTIONS` list.

```
public static var ALL_OPTIONS : List<ProducerWriteOffAgeRestriction> = {LAST_7_DAYS, LAST_30_DAYS, ...}
```

Build and deploy the application. The producer **Write-Offs** screen will include the new seven-day age option.



## chapter 38

# Configuring Credit Handling

This topic describes how to configure credit handling in BillingCenter.

This topic includes:

- “Defining New Return Premium Plan Property Settings” on page 451
- “Defining New Excess Treatment Settings” on page 452
- “Identifying Eligible Invoice Items” on page 454
- “Configuring Credit Allocation” on page 454

**See also**

- “Credit Handling” on page 191 in the *Application Guide*

## Defining New Return Premium Plan Property Settings

A return premium plan includes properties that define how to process a policy period credit. The plan defines properties and each handling scheme in the plan includes additional properties. For details, see “Return Premium Plan Properties” on page 196 in the *Application Guide*.

The available settings for most properties are defined by a typelist associated with the property. New property settings can be defined by adding new typecodes to the property’s typelist. Of course, new program code must also be written to recognize and handle the setting in the desired manner.

The following table lists the properties of return premium plans and handling schemes that can be extended by adding typecodes to their associated typelist.

Property	Associated Typelist
Positive Item Qualifier	ReturnPremiumChargeQualification
List Bill Account Excess	ListBillAccountExcess
Allocate Method	ReturnPremiumAllocateMethod. Adding a new allocation method requires additional steps. See “Defining a New Credit Allocation Method” on page 455.
Allocate Timing	ReturnPremiumAllocateTiming
Context (Handling Condition)	ReturnPremiumHandlingCondition. Adding a new context requires additional steps. Refer to the <code>LinkedImplementationLoaderImpl</code> class and its <code>returnPremiumSchemeIdentificationPredicates</code> method.

Property	Associated Typelist
Excess Treatment	ReturnPremiumExcessTreatment. For examples demonstrating how to implement new methods to process a remaining credit balance, see “Defining New Excess Treatment Settings” on page 452.
Start Date	ReturnPremiumStartDateOption

## Defining New Excess Treatment Settings

In the base configuration, the handling scheme property `Excess Treatment` supports only the `Unapplied` setting. To define additional settings, a new typecode must be added to the `ReturnPremiumExcessTreatment` typelist and program code must be written to recognize and handle the setting in the desired manner. This section provides examples that implement new settings for the `Excess Treatment` property.

### Defining a New Excess Treatment Setting: Disbursement

The following example demonstrates how to define a new `Excess Treatment` setting that creates a disbursement for the remaining credit balance. This handling method is implemented only for credits created because of a policy cancellation.

#### Add a typecode for the new setting

The `ReturnPremiumExcessTreatment` typelist must define a typecode for the new setting. The manner in which the typecode is defined depends on whether the typelist has already been extended.

- If this is the first extension to the `ReturnPremiumExcessTreatment` typelist:
  1. In BillingCenter Studio, in the Project window, navigate to `configuration` → `config` → `Metadata` → `Typelist`.
  2. Right-click `ReturnPremiumExcessTreatment.tti` and choose `New` → `Typelist Extension` and then click `OK`. Studio creates `ReturnPremiumExcessTreatment.ttx` in `configuration` → `config` → `Extensions` → `Typelist` and opens it in the Typelist editor.
- If the `ReturnPremiumExcessTreatment` typelist has already been extended:
  1. In BillingCenter Studio, in the Project window, navigate to `configuration` → `config` → `Extensions` → `Typelist`.
  2. Double-click `ReturnPremiumExcessTreatment.ttx` to open it in the Typelist editor.

Continue with the following steps to define the new typecode.

1. In `ReturnPremiumExcessTreatment.ttx`, right-click an existing typecode and choose `Add new` → `typecode`.
2. Enter the following values for the new typecode:
  - **code** – `Disbursement`
  - **name** – `Disbursement`
  - **desc** – `Disburse any excess credit`

**Note:** Adding or changing a typecode is a data model change, and therefore requires a restart of the application.

#### Write support code to recognize and handle the new setting

The disbursement is created after the credit has been allocated in the `DirectBillPayment` plugin method `allocateCredits`. Append the following code to the end of the method. For information about a particular method, refer to the *Gosu API Reference*.

```
// *** Append to DirectBillPayment.allocateCredits() ***
// Credit must be related to a policy period, not an account or collateral credit
if (policyPeriod != null) {
```

```

var excessTreatment = typekey.ReturnPremiumExcessTreatment.TC_UNAPPLIED // default setting
var moneyReceived = payment.BaseMoneyReceived as DirectBillMoneyRcvd

// If this is a Cancellation credit, retrieve the Excess Treatment setting.
// NOTE: In this example, the Disbursement setting is implemented for Cancellation credits only.
// The Disbursement setting is not implemented for other credit types and will have no effect.
if (moneyReceived.DBPmntDistributionContext == DBPmntDistributionContext.TC_CANCELLATION) {
    excessTreatment = policyPeriod.ReturnPremiumHandlingSchemes.firstWhere( \ elt ->
        elt.HandlingCondition == typekey.ReturnPremiumHandlingCondition.TC_CANCELLATION).ExcessTreatment
}

// If there is remaining credit and it should be disbursed...
if (amountToAllocate.IsNotNull &&
    amountToAllocate.IsPositive &&
    excessTreatment == typekey.ReturnPremiumExcessTreatment.TC_DISBURSEMENT) {
    // Perform the disbursement
    var disbursement = gw.account.CreateDisbursementWizardHelper.createDisbursement(
        policyPeriod.Account, amountToAllocate)
    disbursement = policyPeriod.getBundle().add(disbursement)
    disbursement.executeDisbursementOrCreateApprovalActivityIfNeeded()
}
}

```

The new support code calls a `createDisbursement` method in the `CreateDisbursementWizardHelper` class. This method must be implemented. Add the following code to the class definition.

```

// *** Add to CreateDisbursementWizardHelper class definition ***
static function createDisbursement(account : Account, amount : MonetaryAmount) : AccountDisbursement {
    var disbursement = new AccountDisbursement(account.Currency)
    disbursement.setUnappliedFundsAndFields(account.DefaultUnappliedFund)
    disbursement.Amount = amount
    disbursement.DueDate = gw.api.util.DateUtil.currentDate()
    disbursement.Status = typekey.DisbursementStatus.TC_APPROVED
    disbursement.Reason = typekey.Reason.TC_OVERPAY
    disbursement.Address = account.PrimaryPayer.Contact.PrimaryAddress.toString()
    disbursement.MailTo = account.PrimaryPayer.Contact.toString()
    disbursement.PayTo = account.PrimaryPayer.Contact.toString()
    disbursement.PaymentInstrument = account.DefaultPaymentInstrument
    return disbursement
}

```

## Defining a New Excess Treatment Setting: Suspense Item

The following example demonstrates how to define a new Excess Treatment setting that places the remaining credit balance in a suspense item. This handling method is implemented only for credits created because of a policy cancellation.

### Add a typecode for the new setting

The `ReturnPremiumExcessTreatment` typelist must define a typecode for the new setting. The manner in which the typecode is defined depends on whether the typelist has already been extended.

- If this is the first extension to the `ReturnPremiumExcessTreatment` typelist:
  1. In BillingCenter Studio, in the Project window, navigate to **configuration** → **config** → **Metadata** → **Typelist**.
  2. Right-click `ReturnPremiumExcessTreatment.tti` and choose **New** → **Typelist Extension** and then click **OK**. Studio creates `ReturnPremiumExcessTreatment.ttx` in **configuration** → **config** → **Extensions** → **Typelist** and opens it in the Typelist editor.
- If the `ReturnPremiumExcessTreatment` typelist has already been extended:
  1. In BillingCenter Studio, in the Project window, navigate to **configuration** → **config** → **Extensions** → **Typelist**.
  2. Double-click `ReturnPremiumExcessTreatment.ttx` to open it in the Typelist editor.

Continue with the following steps to define the new typecode.

1. In `ReturnPremiumExcessTreatment.ttx`, right-click an existing typecode and choose **Add new** → **typecode**.
2. Enter the following values for the new typecode:
  - **code** – `SuspenseItem`

- **name** – SuspenseItem
- **desc** – Create Suspense Item

**Note:** Adding or changing a typecode is a data model change, and therefore requires a restart of the application.

### Write support code to recognize and handle the new setting

The suspense item is created after the credit has been allocated in the `DirectBillPayment` plugin method `allocateCredits`. Append the following code to the end of the method. For information about a particular method, refer to the *Gosu API Reference*.

```
// *** Append to DirectBillPayment.allocateCredits() ***
// Credit must be related to a policy period, not an account or collateral credit
if (policyPeriod != null) {
    var excessTreatment = typekey.ReturnPremiumExcessTreatment.TC_UNAPPLIED // default setting
    var moneyReceived = payment.BaseMoneyReceived as DirectBillMoneyRcvd

    // If this is a Cancellation credit, retrieve the Excess Treatment setting.
    // NOTE: In this example, the SuspenseItem setting is implemented for Cancellation credits only.
    // The SuspenseItem setting is not implemented for other credit types and will have no effect.
    if (moneyReceived.DBPmntDistributionContext == DBPmntDistributionContext.TC_CANCELLATION) {
        excessTreatment = policyPeriod.ReturnPremiumHandlingSchemes.firstWhere( \ elt ->
            elt.HandlingCondition == typekey.ReturnPremiumHandlingCondition.TC_CANCELLATION).ExcessTreatment
    }

    // If there is remaining credit and it should be placed in a suspense item...
    if (amountToAllocate.IsNotNull &&
        amountToAllocate.IsPositive &&
        excessTreatment == typekey.ReturnPremiumExcessTreatment.TC_SUSPENSEITEM) {
        // Create a suspense item and place the remaining credit in it
        var item : BaseSuspDistItem
        item = payment.createAndAddSuspDistItem()
        item.GrossAmountToApply = amountToAllocate
        item.MatchingPolicy      = policyPeriod
        item.PolicyNumber        = policyPeriod.PolicyNumber
    }
}
```

## Identifying Eligible Invoice Items

During the credit handling process, BillingCenter filters invoice items to identify and select items eligible to be paid by a credit allocation.

One filter criteria is specified by the `Positive Item Qualifier` property contained in the return premium plan. The base configuration provides several property settings that are described in the section “Positive Item Qualifier” on page 196 in the *Application Guide*.

The filter criteria of the `Positive Item Qualifier` are initialized in the `DirectBillPayment` plugin method `addPositiveItemsDistributionCriteria`. By modifying the implementation of this method, the filter criteria can be customized to meet unique requirements. For additional details, refer to “Customize the Filter Criteria for Credit Allocations” on page 193 in the *Integration Guide*.

## Configuring Credit Allocation

BillingCenter allocates credits using the `DirectBillPayment` plugin. The base configuration calls the plugin method `allocateCredits` to validate the existence of a negative credit and positive invoice items eligible to receive the allocation. Then, based on the type of credit, the method calls the applicable `allocate` method to perform the allocation. The `DirectBillPayment` plugin method `allocateCredits` is described in “Allocate Credits” on page 194 in the *Integration Guide*.

Policy credits, such as return premium credits, are allocated based on the relevant handling scheme for the type of credit being processed. Each handling scheme implements an `allocate` method to perform the allocation. The `allocateCredits` method determines the appropriate handling scheme and calls the scheme's `allocate` method.

The base configuration can be customized by modifying the plugin method `allocateCredits` or by defining a new allocation method and associated `allocate` method.

## Defining a New Credit Allocation Method

To customize the allocation of policy credits, a new `allocate` method must be defined. Each handling scheme defines its own `allocate` method.

**Note:** The `allocate` methods provided in the handling schemes of the base configuration must not be modified. Instead, create a new allocation method as described in this section. The source code from an existing `allocate` method can be copied into the new method to provide a foundation for further development.

Adding an allocation method is a multi-step process:

1. Add a typecode for the new allocation method to the `ReturnPremiumAllocateMethod` typelist.
2. Create a new allocation class that implements `ReturnPremiumAllocationStrategy`. The class requires the following property and method.
  - A `TypeKey` property that returns the new typecode added to the `ReturnPremiumAllocateMethod` typelist.
  - An `allocate` method that accepts a collection of invoice items and allocates the negative credit items to pay the positive items.
3. Instantiate the allocation class so BillingCenter can access it and call its `allocate` method. This is achieved by appending an entry to the list of credit allocation strategies specified in the `LinkedImplementationLoaderImpl.returnPremiumAllocationStrategies` method.

## Defining a New Credit Allocation Method: An Example

In this example, allocation occurs `First to Last`. However, invoice items are grouped into two tiers based on their policy period. Items having the same policy period as the credit are paid first. Afterward, any remaining credit is allocated to items belonging to other policy periods. Items in each group are paid in a `First to Last` manner.

To enable the `DirectBillPayment` plugin `allocateCredits` method to call the new allocation method, the following required steps are performed.

### Add a typecode for the new allocation method

The `ReturnPremiumAllocateMethod` typelist must define a typecode for the new allocation method. The manner in which the typecode is defined depends on whether the typelist has already been extended.

- If this is the first extension to the `ReturnPremiumAllocateMethod` typelist:
  1. In BillingCenter Studio, in the Project window, navigate to `configuration` → `config` → `Metadata` → `Typelist`.
  2. Right-click `ReturnPremiumAllocateMethod.tti` and choose `New` → `Typelist Extension` and then click `OK`. Studio creates `ReturnPremiumAllocateMethod.ttx` in `configuration` → `config` → `Extensions` → `Typelist` and opens it in the Typelist editor.
- If the `ReturnPremiumAllocateMethod` typelist has already been extended:
  1. In BillingCenter Studio, in the Project window, navigate to `configuration` → `config` → `Extensions` → `Typelist`.
  2. Double-click `ReturnPremiumAllocateMethod.ttx` to open it in the Typelist editor.

Continue with the following steps to define the new typecode.

1. In `ReturnPremiumAllocateMethod.ttx`, right-click an existing typecode and choose `Add new` → `typecode`.

**2.** Enter the following values for the new typecode:

- **code** – PolicyPeriodFirst
- **name** – Policy Period First
- **desc** – Use first-to-last allocation. Distribute first to items in the same policy period as the credit item, then to remaining items.

**Note:** Adding or changing a typecode is a data model change, and therefore requires a restart of the application.

**Create a class that implements the ReturnPremiumAllocationStrategy interface**

**1.** Create a class that implements ReturnPremiumAllocationStrategy.

Name the new class `PolicyPeriodFirstReturnPremiumAllocationStrategy`. As a starting development foundation, load the source file that implements the `ProportionalReturnPremiumAllocation` class and copy the implementations of the `TypeKey` property and `allocate` method into the new class.

**Note:** The new class will be registered by `LinkedImplementationLoaderImpl`, therefore it needs to be thread-safe.

**2.** Modify the `TypeKey` property of the new class so it returns the new typecode that was added to the `ReturnPremiumAllocateMethod` typelist.

**3.** Modify the `allocate` method, as follows.

```
override function allocate(distItems: List <BaseDistItem>, amountToAllocate: AllocationPool) {
    // Validate that the required negative credit and positive invoice items exist.
    if (distItems.Empty) {
        return
    }
    var targetDist = distItems[0].BaseDist
    Preconditions.checkNotNull(distItems.where(\elt -> elt.BaseDist != targetDist).size() == 0,
        "All dist items must belong to the same base dist.")

    var positiveDistItems = distItems.where(\distItem -> distItem.InvoiceItem.Amount.IsPositive)
    if (positiveDistItems.Empty) {
        return
    }

    var negativeDistItems = distItems.where(\distItem -> distItem.InvoiceItem.Amount.IsNegative)
    if (negativeDistItems.Empty) {
        return
    }

    // Get the policy period of the first credit item
    var policyPeriod = negativeDistItems[0].PolicyPeriod

    // Get all the positive invoice items that have the same policy period as the credit item.
    // These items comprise the "preferred" group that will receive credit allocations first.
    var preferredPositiveDistItems = positiveDistItems.where(
        \ elt -> elt.PolicyPeriod == policyPeriod)

    // Calculate the amount to allocate to preferred items
    var currency = distItems[0].Currency
    var amountNeededForPreferred = preferredPositiveDistItems.sum(currency,
        \ elt -> elt.GrossAmountOwed - elt.GrossAmountToApply)
    var amountAvailableForPreferred = amountNeededForPreferred.min(amountToAllocate.GrossAmount)

    // Allocate credit to the preferred invoice items based on a First-to-Last allocation method
    new FirstToLastAllocationStrategy().allocate(
        preferredPositiveDistItems, AllocationPool.withGross(amountAvailableForPreferred))

    // Calculate the amount necessary for the remaining items. If any credit remains, allocate it
    // to these "nonpreferred" items. Use a First-to-Last allocation method.
    var amountAvailableForNonPreferred = amountToAllocate.GrossAmount - amountAvailableForPreferred
    if (amountAvailableForNonPreferred.IsPositive) {
        positiveDistItems.removeAll(preferredPositiveDistItems)
        new FirstToLastAllocationStrategy().allocate(positiveDistItems,
            AllocationPool.withGross(amountAvailableForNonPreferred))
    }
}
```

**Instantiate the new allocation class and enable BillingCenter access**

1. In BillingCenter Studio, in the Project window, navigate to configuration → gsrc and then to gw.plugin.system.dependency. Double-click LinkedImplementationLoaderImpl.gs to open the editor.
2. Find the LinkedImplementationLoaderImpl.returnPremiumAllocationStrategies method.
3. To the end of the list of allocation strategies, append a call that instantiates the new allocation class, as shown below.

```
uses PolicyPeriodFirstReturnPremiumAllocationStrategy

override function returnPremiumAllocationStrategies() : Collection<ReturnPremiumAllocationStrategy> {
    return {
        new FirstToLastReturnPremiumAllocationStrategy(),
        new LastToFirstReturnPremiumAllocationStrategy(),
        new ProportionalReturnPremiumAllocationStrategy(),
        new PolicyPeriodFirstReturnPremiumAllocationStrategy()
    }
}
```

**Note:** All classes registered by LinkedImplementationLoaderImpl must be thread-safe.

4. Make and deploy the project.





## chapter 39

# Configuring Multicurrency

This topic includes:

- “Configuring BillingCenter with a Single Currency” on page 459
- “Configuring BillingCenter with Multiple Currencies” on page 459
- “Enabling Multicurrency Integration” on page 461

**See also**

- “Multicurrency Overview” on page 269 in the *Application Guide*
- “Configuring Currencies” on page 113 in the *Globalization Guide*

## Configuring BillingCenter with a Single Currency

To configure BillingCenter to use a single currency, set the default application currency and the multicurrency display mode parameters in `config.xml`.

For details see:

- “Setting the Default Application Currency” on page 119 in the *Globalization Guide*
- “Currency-related Configuration Parameters” on page 119 in the *Globalization Guide*

## Configuring BillingCenter with Multiple Currencies

To configure BillingCenter to use multiple currencies, set the default application currency and the multicurrency display mode parameters in `config.xml`. The default currency populates the `Currency` property for new accounts and producers unless you explicitly specify another currency.

For details see:

- “Setting the Default Application Currency” on page 119 in the *Globalization Guide*
- “Currency-related Configuration Parameters” on page 119 in the *Globalization Guide*

## Currency Silos

You can use multiple currencies in the same instance of BillingCenter, but each financial entity and all of its related entities must share the same currency. BillingCenter never transfers or converts money from one currency to another. BillingCenter never compares two monetary amounts in different currencies.

One way to keep currencies separate is to use separate databases and separate instance of BillingCenter. This approach is too restrictive.

Instead, BillingCenter prevents monetary amounts in different currencies from mixing by allowing each entity to use only one currency. Then BillingCenter allows connections between entities only if the entities use the same currency.

In multicurrency mode, BillingCenter behaves essentially like multiple separate single-currency instances that share different parts of the same database. Rather than require a separate database for each currency, BillingCenter simulates separate database instances for each currency with *currency silos*.

Currency silos create barriers that prevent the co-mingling of currencies at the database level. Each currency has a silo. Entities in one currency silo can only be associated with other entities in the same silo. A transaction can not interact with entities in more than one silo. Currency silos are like virtual databases within the single BillingCenter instance.

There are a few exceptions to the restriction that entities can only associate with other entities in the same silo. BillingCenter does allow some associations across currency silos. For example, a parent account can have an assigned currency that is different from any of its child accounts. You can model a policyholder with policies in multiple different currencies using a parent account with a child account owner for each currency.

Similarly, a producer has an assigned currency and all associated plans and policies must share the same currency as the producer. To model a producer that handles policies in multiple currencies you must create a separate BillingCenter producer for each currency.

## Multicurrency Delegates

All financial BillingCenter entities must delegate to `InCurrencySilo`. All other BillingCenter entities must delegate to `NotInCurrencySilo`. Only those entities that delegate to `NotInCurrencySilo` can hold data associated with more than one currency.

For example:

```
<entity entity="Account" ... >
  <implementsEntity name="InCurrencySilo"/>
  ...
</entity>
<entity entity="AuthorityLimit" ... >
  <implementsEntity name="NotInCurrencySilo"/>
  ...
</entity>
```

Attempting to instantiate `InCurrencySilo` entities without providing a currency generates a compile time error.

## Monetary Data Types

The `MonetaryAmount` data type represents money in BillingCenter. This data type corresponds to two columns in the database: one representing the amount and another representing the currency. See “Monetary Amounts in the Data Model and in Gosu” on page 115 in the *Globalization Guide* for details.

---

**WARNING** Do not use the `currencyamount` column type in your data model nor the corresponding `CurrencyAmount` Gosu or Java type.

---

## Exceptions to Currency Silo Rules

If your configuration requires you to connect entities across silos, you can exclude a specific property from the currency silo. For example, you want to add a property of type `Currency` to an entity that is in a currency silo, but allow that property to use any currency. Essentially, you want to exclude that property from the currency silo. Because you are not benefitting from the protection that currency silos provide, you must prevent monetary amounts from being mixed across currencies.

Any property of type `Currency` or `MonetaryAmount` is excluded from the silo if it is defined with the tag `ExcludeFromSilo`. For example:

```
<typekey name="ExcludedCurrency"
         desc="Currency column that is excluded from the silo."
         typelist="Currency">
    <tag name="ExcludeFromSilo"/>
</typekey>
<monetaryamount
         name="ExcludedMonetaryAmount"
         desc="MonetaryAmount that is excluded from the silo.">
    <tag name="ExcludeFromSilo"/>
</monetaryamount>
```

You may also want to allow some links between silos. As with excluding properties from the silo, for any associations across silos you allow, you are forgoing the protection that currency silos provide. Therefore your configuration code must protect against mixing currencies between the connected silos.

Any foreign key property is excluded from the silo if it is defined with the tag `CanLinkToAnotherSilo`. For example:

```
<foreignkey fkentity="AnotherSiloedEntity"
            name="CrossSiloLink"
            desc="Link to another entity that can be in another silo.">
    <tag name="CanLinkToAnotherSilo"/>
</foreignkey>
```

## Enabling Multicurrency Integration

You must perform certain tasks to enable multicurrency integration appropriately.

- “Set up Currency-Specific Plans and Authority Limits” on page 461

### Set up Currency-Specific Plans and Authority Limits

One of the administrative tasks you must perform when setting up BillingCenter is to create plans. Several of these plans are currency specific, and therefore you must set up at least one plan for each currency. There are monetary values that reflect various applicable fees and financial thresholds. For example, billing plans contain an invoice fee. A Euro account must define the fee in euros. As another example, delinquency plans have monetary thresholds that control when BillingCenter makes a policy delinquent. Such thresholds must be defined for each currency.

These currency specific administrative entities that need to be created in BillingCenter are:

- Billing plans
- Payment plans
- Commission plans
- Delinquency plans
- Agency bill plans
- Authority limits

BillingCenter requires all of these be defined to issue a policy in a given currency, except agency bill plans. You must define agency bill plans for each currency in which you plan on issuing policies with agency billing. If you do not define an agency bill plan for a given currency, **Agency Bill** will not appear in PolicyCenter as a choice for billing method.

**See also**

- “Plan Types” on page 101 in the *Application Guide*

### [Set up Billing Plans in BillingCenter for Multicurrency Accounts](#)

In BillingCenter, each account must have an associated billing plan. Billing plans determine how BillingCenter handles automatic distributions, special circumstances such as low balance, and invoicing at the account level. Billing plans are currency specific and therefore you must set up at least one billing plan in BillingCenter for each currency.

**See also**

- “Working with Billing Plans” on page 110 in the *Application Guide*

### [Set up Payment Plans in BillingCenter for Multicurrency Accounts](#)

In BillingCenter, each policy must have an associated payment plan. Payment plans determine how payments are set up and spread over the term of the policy. Payment plans are currency specific and therefore you must set up at least one billing plan in BillingCenter for each currency. The payment plans that match the currency of a policy appear in PolicyCenter as a list of choices for how to bill a policy.

**See also**

- “Working with Payment Plans” on page 119 in the *Application Guide*

### [Set up Agency Bill Plans in BillingCenter for Multicurrency Producer Organizations](#)

In PolicyCenter you set the currency for producer organizations through agency bill plans. PolicyCenter retrieves the agency bill plans from BillingCenter. In a single currency system, you select an **Agency Bill Plan** on the **Basics** tab of the **Organization** screen. In a multicurrency system, you select one or more agency bill plans on the **Agency Bill** tab. Each agency bill plan specifies a currency.

When entering the **Organizations** or the **New Organizations** screens, PolicyCenter synchronizes with BillingCenter to refresh the agency bill plans.

**See also**

- “Working with Agency Bill Plans” on page 136 in the *Application Guide*
- “Organizations and Producer Codes Overview” in the *PolicyCenter Application Guide*

### [Set up Delinquency Plans in BillingCenter for Multicurrency Accounts](#)

A BillingCenter delinquency plan defines a sequence of events to execute when a policy becomes past due. The delinquency plan can be applied to the past-due policy only or to all policies in the account.

A special case exists when all policies in the account are considered delinquent and the account supports multiple currencies. In this situation, the delinquency plan is applied only to the policies that use the same currency as the policy that initiated the delinquency. In other words, only policies contained in the initiating policy’s currency silo are considered to be delinquent. Other policies in the account that use a different currency (and are therefore in a different currency silo) are not considered delinquent. This behavior describes the operation of the BillingCenter base configuration. If you wish all policies in the multicurrency account to be considered delinquent, regardless of their currency, BillingCenter can be configured to support that behavior.

Similarly, when a delinquent account is reinstated, the base configuration reinstates only the policies that reside in the relevant currency silo. To reinstate all policies in the account, regardless of their currency, configuration can be written to support that behavior.

**See also**

- “Working with Delinquency Plans” on page 128 in the *Application Guide*

## [Set up Commission Plans in BillingCenter in for Multicurrency Producer Codes](#)

In PolicyCenter, producer codes are associated with commission plans. Each commission plan is associated with a currency. PolicyCenter retrieves the commission plans from BillingCenter. In a single currency system, each producer code has a single commission plan. In a multicurrency system, a producer code must have multiple commission plans, one for each currency.

When entering the **Producer Codes** or the **New Producer Code** screens, PolicyCenter synchronizes with BillingCenter to refresh the commission plans for the current producer code.

**See also**

- “Working with Commission Plans” on page 151 in the *Application Guide*
- “Organizations and Producer Codes Overview” in the *PolicyCenter Application Guide*

## [Set up Authority Limits in BillingCenter for Multicurrency Integration](#)

Each user in BillingCenter needs to be associated to an authority limit profile. BillingCenter supports multicurrency Authority Limit Profiles. Authority Limit Profiles are composed of authority limits, and an authority limit must be defined separately for each currency. For example, one authority limit specifies the maximum amount a user has the authority to write off. If your integration includes Japanese yen, European Union euros, and U.S. dollars, a separate write-off authority limit must be created for each of these currencies. All three of these currency specific authority limits are associated to the same authority limit profile.

**See also**

- “Authority Limits and Authority Limit Profiles” on page 386 in the *Application Guide*

**See also**

- “Multicurrency Integration between BillingCenter and PolicyCenter” on page 405 in the *Integration Guide*

