

Guidewire BillingCenter®

BillingCenter Upgrade 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 Upgrade Guide

Document Revision: 02-July-2015

Contents

About BillingCenter Documentation	11
Conventions in This Document	12
Support	12

Part I

Planning the Upgrade

1 Planning Your BillingCenter Upgrade	15
Supported Starting Version	16
Upgrading from Version 2.1	16
Upgrading Language Packs	17
Roadmap for Planning the Upgrade	17
Upgrade Assessment	17
Preparing for the Upgrade	19
Project Inception	20
Design and Development	21
System Test	21
Deployment and Support	21
Sample Deployment Plan	21

Part II

Upgrading from 8.0.x

2 Upgrading the BillingCenter 8.0.x Configuration	29
Overview of ContactManager Upgrade	30
Obtaining Configurations and Tools	30
Viewing Differences Between Base and Target Releases	31
Specifying Configuration and Tool Locations	31
Creating a Configuration Backup	34
Removing Patches	34
Removing Language Packs	34
Updating Infrastructure	34
Launching the BillingCenter 8.0.4 Configuration Upgrade Tool	35
Restarting the Configuration Upgrade Tool	35
Configuration Upgrade Tool Automated Steps	36
Copying Custom Rules and Adding BillingCenter 8.0.4 Default Rules	36
Using the BillingCenter 8.0.4 Upgrade Tool Interface	36
Filters	37
Configuration File Tree	41
File Details Panel	41
Accepting Files that Do Not Require Merging	42
Merging and Accepting Files	42
Configuration Merging Guidelines	42

Data Model Merging Guidelines	43
Merging Typelists – Overview	43
Merging Typelists – Simple Typelists	44
Merging Typelists – Complex Typelists.....	44
Reviewing Shared Typekey Configuration.....	44
Merging Entity Extensions	44
Reviewing Custom Extensions	45
Reconciling the Database with Custom Extensions	45
Merging Display Properties	45
Upgrading Rules to BillingCenter 8.0.4	46
Translating New Display Properties and Typecodes	47
Validating the BillingCenter 8.0.4 Configuration	47
Using Studio to Verify Files	47
Starting BillingCenter and Resolving Errors	48
Building and Deploying BillingCenter 8.0.4	48
3 Upgrading the BillingCenter 8.0.x Database	51
Upgrading Administration Data for Testing.....	52
Identifying Data Model Issues	53
Verifying Batch Process and Work Queue Completion.....	54
Purging Data Prior to Upgrade	54
Purging Old Messages from the Database	54
Purging Completed Workflows and Workflow Logs	55
Validating the Database Schema	55
Checking Database Consistency.....	56
Creating a Data Distribution Report.....	56
Generating Database Statistics	57
Creating a Database Backup.....	58
Updating Database Infrastructure	58
Preparing the Database for Upgrade.....	58
Ensuring Adequate Free Space	58
Disabling Replication	58
Assigning Default Tablespace (Oracle only)	58
Setting Linguistic Search Collation	59
Field Encryption and the Upgraded Database	60
Customizing the Upgrade	60
Running Custom Version Checks and Triggers	61
IDatamodelUpgrade API Examples	64
Running the Commission Payable Calculations Process	71
Configuring the Database Upgrade.....	71
Adjusting Commit Size for Encryption	72
Configuring Version Trigger Elements.....	72
Deferring Creation of Nonessential Indexes.....	73
Configuring the Upgrade on Oracle	74
Configuring the Upgrade on SQL Server	76
Downloading Database Upgrade Instrumentation Details	77
Checking the Database Before Upgrade.....	78
Disabling the Scheduler	78
Suspending Message Destinations	79
Starting the Server to Begin Automatic Database Upgrade	79
Test the Database Upgrade	79
Integrations and Starting the Server	80
Understanding the Automatic Database Upgrade.....	80
Version Trigger Descriptions	81

Viewing Detailed Database Upgrade Information	83
Dropping Unused Columns on Oracle	83
Exporting Administration Data for Testing	84
Final Steps After The Database Upgrade is Complete	85
Checking that Contacts Have Unique Addresses	86
Completing Deferred Upgrade	86
Reenabling Database Logging	86
Backing up the Database After Upgrade.....	86
4 Upgrading BillingCenter from 8.0.x for ContactManager	87
Manually Upgrading BillingCenter to Integrate with ContactManager	87
File Changes in BillingCenter Related to ContactManager.	88
Web Service Version Changes	88
5 Upgrading ContactManager from 8.0.x.....	89
Manually Upgrading the ContactManager Configuration	89
Manually Configuring Changed Files.....	90
BillingCenter Web Services Version Change.....	90

Part III Upgrading from 7.0.x

6 Upgrading the BillingCenter 7.0.x Configuration.....	95
Overview of ContactManager Upgrade	96
Obtaining Configurations and Tools.	96
Viewing Differences Between Base and Target Releases	97
Specifying Configuration and Tool Locations	97
Creating a Configuration Backup	100
Removing Patches.....	100
Removing Language Packs.....	100
Updating Infrastructure.....	100
Launching the BillingCenter 8.0.4 Configuration Upgrade Tool.....	101
Restarting the Configuration Upgrade Tool	101
Configuration Upgrade Tool Automated Steps	102
Removing Template Pages	102
Updating PCF Files.....	102
Upgrading Work Queue Configuration.....	104
Upgrading Database Configuration.	104
Splitting Localization.xml into Separate Files for each Locale	107
Splitting address-config.xml into Separate Files for each Country	107
Splitting zone-config.xml into Separate Files for each Country.....	107
Splitting currencies.xml into Separate Files for each Currency	107
Moving Country-based Field Validator Definition Files	107
Moving Rules Files up One Directory	107
Reformatting Rules for Display in Studio Rules Editor	107
Copying Custom Rules and Adding BillingCenter 8.0.4 Default Rules.....	107
Renaming SOAP Web Services from XML to RWS	108
Renaming Plugins from XML to GWP.....	108
Renaming Display Names Files from XML to EN.....	108
Upgrading Display Keys.....	108
Adding nullok="true" to Entity and Extension Foreign Key Columns.....	108
Removing deletefk Attribute from Entity and Extension Foreign Keys.....	108
Setting XML Namespace on Metadata Files	108
Upgrading Document Assistant Parameters	109
Separating Entities and Typelists	109

Using the BillingCenter 8.0.4 Upgrade Tool Interface	109
Filters	110
Configuration File Tree	114
File Details Panel	114
Accepting Files that Do Not Require Merging	115
Merging and Accepting Files	115
Configuration Merging Guidelines	115
Data Model Merging Guidelines	116
Updating Data Types for Case Sensitivity	116
Merging Typelists – Overview	116
Merging Typelists – Simple Typelists	117
Merging Typelists – Complex Typelists	117
Reviewing Shared Typekey Configuration	117
Adding State Typelist Extensions to Jurisdiction	118
Merging Entity Extensions	118
Reviewing Custom Extensions	119
Reconciling the Database with Custom Extensions	119
Removing Obsolete Attributes	119
Updating Extractable Edge Foreign Keys	119
Converting Money to MonetaryAmount	120
Changes to the Logging API	121
Conceptual Changes to Logging	121
Instantiating Loggers	122
Logging Messages	123
Passing Loggers as Parameters	123
Adding DDL Configuration Options to database-config.xml	124
Merging Changes to Field Validators	124
Renaming PCF files According to Their Modes	125
Updating Rounding Mode Parameter	125
Merging Display Properties	125
Merging Other Files	126
Fixing Gosu Issues	126
Gosu Case Sensitivity	126
Inequality Operator	127
Ambiguous Method Calls	127
Nested Comments	128
Upgrading Rules to BillingCenter 8.0.4	128
Translating New Display Properties and Typecodes	129
Validating the BillingCenter 8.0.4 Configuration	130
Using Studio to Verify Files	130
Starting BillingCenter and Resolving Errors	130
Building and Deploying BillingCenter 8.0.4	131
7 Upgrading the BillingCenter 7.0.x Database	133
Upgrading Administration Data for Testing	134
Identifying Data Model Issues	135
Verifying Batch Process and Work Queue Completion	136
Purging Data Prior to Upgrade	136
Purging Old Messages from the Database	136
Purging Completed Workflows and Workflow Logs	137
Validating the Database Schema	137
Checking Database Consistency	138
Creating a Data Distribution Report	138
Generating Database Statistics	139

Creating a Database Backup	140
Updating Database Infrastructure	140
Preparing the Database for Upgrade	140
Ensuring Adequate Free Space	140
Disabling Replication	140
Assigning Default Tablespace (Oracle only)	140
Setting Linguistic Search Collation	141
Field Encryption and the Upgraded Database	142
Customizing the Upgrade	142
Running Custom Version Checks and Triggers	143
IDatamodelUpgrade API Examples	146
Running the Commission Payable Calculations Process	153
Configuring the Database Upgrade	153
Adjusting Commit Size for Encryption	154
Configuring Version Trigger Elements	154
Deferring Creation of Nonessential Indexes	155
Configuring the Upgrade on Oracle	156
Configuring the Upgrade on SQL Server	158
Downloading Database Upgrade Instrumentation Details	159
Checking the Database Before Upgrade	160
Disabling the Scheduler	160
Suspending Message Destinations	161
Starting the Server to Begin Automatic Database Upgrade	161
Test the Database Upgrade	161
Integrations and Starting the Server	162
Understanding the Automatic Database Upgrade	162
Version Trigger Descriptions	163
Viewing Detailed Database Upgrade Information	182
Dropping Unused Columns on Oracle	183
Exporting Administration Data for Testing	184
Upgrading Phone Numbers	185
Final Steps After The Database Upgrade is Complete	186
Completing Deferred Upgrade	187
Reenabling Database Logging	187
Checking that Contacts Have Unique Addresses	187
Backing up the Database After Upgrade	187
8 Upgrading BillingCenter from 7.0.x for ContactManager	189
Configuration File Changes in BillingCenter	190
Manually Upgrading BillingCenter to Integrate with ContactManager	191
Mapping Your Contact Extensions	191
Parameter transactionId Removed from ContactManager Web Services	192
9 Upgrading ContactManager from 7.0.x	193
Database Upgrade Steps in ContactManager	193
Preserving MatchSetKey Column Data	193
Ensuring that LinkID Is Unique	194
Configuration File Changes in ContactManager	194
Manually Configuring Changed Files	195

Part IV

Upgrading from 3.0.x

10 Upgrading the BillingCenter 3.0.x Configuration.....	203
Obtaining Configurations	204
Viewing Differences Between Base and Target Releases	205
Specifying Configuration Locations for BillingCenter 7.0 Upgrade Tool	205
Creating a Configuration Backup	208
Removing Patches.....	208
Removing Language Packs.....	208
Updating Infrastructure.....	208
Upgrading the BillingCenter 3.0 Configuration to 7.0	208
Launching the BillingCenter 7.0 Configuration Upgrade Tool	208
Restarting the Configuration Upgrade Tool	209
BillingCenter 7.0 Upgrade Tool Automated Steps.....	209
Moving Typelist Localizations into typelist.properties Files	209
Removing Redundant TTX Files	209
Removing searchTypeVisible Attribute from DateCriterionChoiceInputNode	210
Copying Display Properties Files into Target Configuration	210
Copying Custom Rules and Adding BillingCenter 8.0.4 Default Rules.....	210
Referencing XSD Files	210
Removing AdminTable Delegate from Custom Extensions.....	210
Converting sessiontimeoutsecs Security Element to Parameter	210
Removing Redundant Batch Server Parameter.....	210
Configuring the BillingCenter 8.0 Upgrade Tool.....	211
Launching the BillingCenter 8.0 Configuration Upgrade Tool	213
Restarting the Configuration Upgrade Tool	214
BillingCenter 8.0.4 Configuration Upgrade Tool Automated Steps	214
Removing Template Pages	214
Updating PCF Files.....	214
Upgrading Work Queue Configuration.....	216
Upgrading Database Configuration.....	217
Splitting Localization.xml into Separate Files for each Locale	219
Splitting address-config.xml into Separate Files for each Country	219
Splitting zone-config.xml into Separate Files for each Country.....	219
Splitting currencies.xml into Separate Files for each Currency	219
Moving Country-based Field Validator Definition Files	219
Moving Rules Files up One Directory	219
Reformatting Rules for Display in Studio Rules Editor	219
Copying Custom Rules and Adding BillingCenter 8.0.4 Default Rules.....	219
Renaming SOAP Web Services from XML to RWS	220
Renaming Plugins from XML to GWP.....	220
Renaming Display Names Files from XML to EN.....	220
Upgrading Display Keys	220
Adding nullok="true" to Entity and Extension Foreign Key Columns.....	220
Removing deletefk Attribute from Entity and Extension Foreign Keys.....	220
Setting XML Namespace on Metadata Files	221
Upgrading Document Assistant Parameters	221
Separating Entities and Typelists	221

Using the BillingCenter 8.0.4 Upgrade Tool Interface	221
Filters	222
Configuration File Tree	226
File Details Panel	226
Accepting Files that Do Not Require Merging	227
Merging and Accepting Files	227
Configuration Merging Guidelines	227
Data Model Merging Guidelines	228
Updating Data Types for Case Sensitivity	228
Merging Typelists – Overview	228
Merging Typelists – Simple Typelists	229
Merging Typelists – Complex Typelists	229
Reviewing Shared Typekey Configuration	229
Adding State Typelist Extensions to Jurisdiction	230
Merging Entity Extensions	230
Reviewing Custom Extensions	231
Reconciling the Database with Custom Extensions	231
Removing Obsolete Attributes	231
Updating Extractable Edge Foreign Keys	231
Converting Money to MonetaryAmount	232
Preserving Payment Method Details	233
Migrating BCContact and PaymentDetails Extensions	235
Changes to the Logging API	235
Conceptual Changes to Logging	236
Instantiating Loggers	237
Logging Messages	237
Passing Loggers as Parameters	237
Changes to Iterators in PCF Files	238
Updating Namespace on Files Loaded by GX Models	239
Adding DDL Configuration Options to database-config.xml	239
Merging Changes to Field Validators	239
Renaming PCF files According to Their Modes	240
Updating Rounding Mode Parameter	240
Merging compatibility-xsd.xml	240
Merging Display Properties	241
Merging Other Files	242
Migrating to 64-bit IDs During Upgrade (SQL Server Only)	242
Fixing Gosu Issues	243
Gosu Case Sensitivity	243
Inequality Operator	243
Ambiguous Method Calls	244
Nested Comments	244
Upgrading Rules to BillingCenter 8.0.4	245
Running PCF Iterator Upgrade	246
Translating New Display Properties and Typecodes	246
Validating the BillingCenter 8.0.4 Configuration	247
Using Studio to Verify Files	247
Starting BillingCenter and Resolving Errors	247
Building and Deploying BillingCenter 8.0.4	248
11 Upgrading the BillingCenter 3.0.x Database	249
Upgrading Administration Data for Testing	250
Identifying Data Model Issues	251
Verifying Batch Process and Work Queue Completion	252

Purging Data Prior to Upgrade	252
Purging Old Messages from the Database	252
Purging Completed Workflows and Workflow Logs	253
Validating the Database Schema	253
Checking Database Consistency	254
Creating a Data Distribution Report	254
Generating Database Statistics	255
Creating a Database Backup	256
Updating Database Infrastructure	256
Preparing the Database for Upgrade	256
Ensuring Adequate Free Space	256
Disabling Replication	256
Assigning Default Tablespace (Oracle only)	256
Setting Linguistic Search Collation	257
Field Encryption and the Upgraded Database	258
Customizing the Upgrade	258
Running Custom Version Checks and Triggers	259
IDatamodelUpgrade API Examples	262
Running the Commission Payable Calculations Process	269
Configuring the Database Upgrade	269
Adjusting Commit Size for Encryption	270
Configuring Version Trigger Elements	270
Deferring Creation of Nonessential Indexes	271
Configuring the Upgrade on Oracle	272
Configuring the Upgrade on SQL Server	274
Downloading Database Upgrade Instrumentation Details	275
Checking the Database Before Upgrade	276
Disabling the Scheduler	276
Suspending Message Destinations	277
Starting the Server to Begin Automatic Database Upgrade	277
Test the Database Upgrade	277
Integrations and Starting the Server	278
Understanding the Automatic Database Upgrade	278
Version Trigger Descriptions	279
Viewing Detailed Database Upgrade Information	314
Dropping Unused Columns on Oracle	315
Exporting Administration Data for Testing	315
Upgrading Phone Numbers	317
Final Steps After The Database Upgrade is Complete	318
Checking that Contacts Have Unique Addresses	318
Completing Deferred Upgrade	319
Reenabling Database Logging	319
Migrating to 64-bit IDs After Upgrade (SQL Server Only)	319
Backing up the Database After Upgrade	319
12 Upgrading Integrations and Gosu from 3.0.x	321
Overview of Upgrading Integration Plugins and Code	321
Tasks Required Before Starting the Server	322
Tasks Required Before Deploying a Production Server	324
Tasks Required Before the Next Upgrade	324

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

Support

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

part I

Planning the Upgrade

Planning Your BillingCenter Upgrade

IMPORTANT Guidewire recommends that you consult with Guidewire Services before beginning an upgrade. Guidewire Services has a number of Knowledge Base articles and Accelerators that might assist with your upgrade.

Upgrade your BillingCenter installation frequently, in order to:

- Incorporate and use the new features of the new release.
- Reduce the number of versions to which you must upgrade to reach the current version.
- Remain compliant with your software license agreement.
- Continue to receive critical product support.

Your existing BillingCenter installation is uniquely configured to meet the specific needs of your business. Thus, it is not possible to fully automate the upgrade process. Guidewire has taken steps to automate as much as possible, but there are always manual activities involved.

This topic assists you in planning a BillingCenter upgrade to version 8.0.4. Read this topic before beginning an upgrade.

Also review the following topics before beginning the upgrade procedure:

- “What’s New and Changed in 8.0.0” on page 33 in the *New and Changed Guide*.
- “What’s New and Changed in 7.0.0” on page 91 in the *New and Changed Guide*.
- “Release Notes Archive” on page 141 in the *New and Changed Guide* for changes to maintenance releases between major versions.
- “Upgrade Issues” in the BillingCenter 8.0.4 release notes for issues specific to this release. If there are no upgrade issues specific to BillingCenter 8.0.4, there is not an “Upgrade Issues” topic in the release notes.
- If you have any language packs installed, see “Upgrading Display Languages” on page 30 in the *Globalization Guide*.

The upgrade procedure is presented in three topics: upgrading the configuration, upgrading the database, and upgrading Gosu code and integration points.

Upgrade topics are presented together in parts of this guide according to your starting version.

If upgrading BillingCenter 8.0, see the topics in “Upgrading from 8.0.x” on page 27.

If upgrading BillingCenter 7.0, see the topics in “Upgrading from 7.0.x” on page 93.

If upgrading BillingCenter 3.0, see the topics in “Upgrading from 3.0.x” on page 201.

This topic includes:

- “Supported Starting Version” on page 16
- “Upgrading Language Packs” on page 17
- “Roadmap for Planning the Upgrade” on page 17
- “Upgrade Assessment” on page 17
- “Preparing for the Upgrade” on page 19
- “Project Inception” on page 20
- “Design and Development” on page 21
- “System Test” on page 21
- “Deployment and Support” on page 21

Supported Starting Version

You can upgrade to BillingCenter 8.0.4 directly from any BillingCenter 3.0 or 7.0 version.

If you are on a 3.0 version prior to 3.0.8, Guidewire recommends that you first upgrade to 3.0.8 and then run consistency checks before continuing the upgrade to 8.0. A direct upgrade from a 3.0 version prior to 3.0.8 can perform poorly, so for the best upgrade performance, upgrade to 3.0.8 first. If you are on a 3.0 version prior to 3.0.8, contact Guidewire Support for a data fix for incorrect commission redistributions.

If you are on a 2.1 version, first upgrade your database to 7.0 using the instructions in the BillingCenter 7.0 *Upgrade Guide*. Then proceed with the upgrade from 7.0 to 8.0 using the instructions in this guide.

Upgrading from Version 2.1

To upgrade your database to 7.0, merge your data model files (entities, typelists, field validators and data types), and correct errors that prevent the 6.0 server from starting. For the production deployment execution, you only need a 7.0 batch server and a 7.0-compatible database. See the *Guidewire Platform Support Matrix* for 7.0 database requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

Since the 7.0 application will never be used interactively by any users, you do not need to upgrade your entire configuration and integrations to 7.0, only the data model.

It is not necessary to upgrade your entire custom configuration to 7.0 and then test the entire application in 7.0 before upgrading to 8.0. This process would add several months to the project duration with no significant benefit.

To upgrade to 7.0

1. Upgrade all data model files (entities, typelists, field validators, data types).

For **BOTH_ADD** files – Merge Guidewire changes with your custom changes, generally keeping both sets of changes.

For **BOTH_EDIT** files – Merge Guidewire changes with your custom changes, generally keeping both sets of changes.

For **EDIT_DELETE** files – Keep your custom version of the file, but make note of the files that are in this category. These files appear under the **CUSTOMER_ADD** filter during the 8.0 merge process. You will need to reevaluate the differences and decide whether to accept the deletion of the file when merging into 8.0.

For all other categories of files – Accept all Guidewire changes and accept all your custom changes.

2. Upgrade all other (non data model) files:

For **BOTH_ADD** files – Keep only your custom version of the file.

For **BOTH_EDIT** files – Keep only your custom version of the file.

For **EDIT_DELETE** files – Keep your custom version of the file, but make note of the files that are in this category. These files appear under the **CUSTOMER_ADD** files during the 8.0 merge process. In most cases, you will not want to accept the file into the 8.0 configuration. Instead, you will rebuild the desired logic in the appropriate files for the 8.0 release.

For all other categories of files – Accept all Guidewire changes and accept all your custom changes.

The process described above will minimize the effort associated with upgrading the database to 7.0, because only the data model files must be merged. This is the minimum work necessary to be able to start the 7.0 server and trigger the automated database upgrade to 7.0. All remaining (non data model) files need only be merged once, into the 8.0 release.

Upgrading Language Packs

This document does not provide instructions for upgrading language packs. See “Upgrading Display Languages” on page 30 in the *Globalization Guide* before continuing with the BillingCenter upgrade. Also see the release notes included with the target release version of the language pack.

IMPORTANT If your base release has a language pack installed, you must uninstall the language pack prior to upgrading.

Roadmap for Planning the Upgrade

Before you begin an upgrade, plan and prepare for how an upgrade impacts both the business processes that rely on your Guidewire product and your organization as a whole. An upgrade requires commitment of time, personnel, and coordination among departments within your company.

Include these steps while defining your upgrade project:

- **Upgrade Assessment** – to plan the upgrade project.
- **Preparing for the Upgrade** – to prepare the environment and people for the upgrade project.
- **Project Inception** – to define the upgrade project.
- **Design and Development** – to implement the changes defined in scope for the upgrade project.
- **System Test** – to perform system, performance and regression testing for the upgrade, as well as perform deployment dry runs.
- **Deployment and Support** – to migrate the upgrade to production and provide necessary post-implementation support during the first few weeks after deployment.

Upgrade Assessment

The first step in planning an upgrade is to determine the impact of the upgrade on your implementation. During this phase of the project, you are:

- Performing an Opportunity Assessment.
- Analyzing the Impact on Your Installation.

- Creating Project Estimates.

These steps provide your users with business benefits and provide a clear understanding of resource requirements and the schedule you need to complete the project.

This phase typically take two to four weeks.

Performing an Opportunity Assessment

The *BillingCenter New and Changed Guide* and the release notes describe new features available after upgrading BillingCenter. As you review the new features, consider:

- How you might leverage these features into business benefits.
- How these features might help you improve your business processes.

Guidewire can also provide you with an evaluation copy of the new release, demonstrate new features for you, and help you understand opportunities these features can create for your business. Install the target version in a test environment. Take time to use the product to discover how to take advantage of the new features. Then, run some common scenarios for your company.

Analyzing the Impact on Your Installation

An upgrade might also impact your existing infrastructure, application configuration, and integration to other software. Make a careful evaluation of:

- Infrastructure Impacts
- Configuration Impacts
- Integration Impacts

Infrastructure Impacts

You might find it desirable or necessary to upgrade your hardware or software. If a major infrastructure element changes, such as a database version, factor that into your upgrade planning.

See the *Guidewire Platform Support Matrix* for system and patch level requirements for BillingCenter 8.0.4. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

Complete any infrastructure updates for your upgrade development environment before you begin the upgrade.

Configuration Impacts

Your company has uniquely configured BillingCenter to meet its particular business needs. During an upgrade, configuration migration from the current version to the target version is your responsibility. Guidewire attempts to assist in this process by providing tools that locate and report back unique configurations in an installation. Additionally, The *BillingCenter New and Changed Guide* and the release notes provide a reference for changes between each version and how these changes impact an installation.

Configuration migrations can take from days to weeks, depending on the complexity of the installation and resources available to perform the migration. If you choose to deploy new features in a release, implementing them can also impact the migration duration.

Pay special attention to areas in which your installation has significant custom configurations. For example, if you have extensively configured a user interface page, review its related files, data objects, and upgrade documentation for specific changes. Guidewire documentation is searchable. Search for key words or attributes to see if they changed between releases. While you do your review, keep in mind the following information:

- Is there current functionality your company uses that is absent from the target version?
- Is there new functionality in the target version that your company will use?
- Do you need to customize the new functionality or is it adequate as provided?

- If you have existing integrations, how are they impacted?
- Are there data changes between the current version and the target version?

Guidewire recommends first running the automated upgrade procedure in a development environment. This can help identify areas requiring work and give an indication of how much work is involved. These areas include:

Data Model – The BillingCenter data model includes all BillingCenter data entities and their relationships. The base data model changes between versions. If you have added extensions to objects in the base data model, the upgrade procedure migrates your extensions. However, if you have rules or integration code that reference your extensions, you are responsible for ensuring that they are correct after the upgrade.

User Interface – The automated process updates user interface customizations if possible. The update tool reports back which files it could not change. You are responsible for identifying unique customizations and migrating them into the new interface.

Other Configuration Files – In addition to user interface configurations, your installation probably includes unique system settings, security settings, and other configuration file settings. Generally, the upgrade preserves customizations to these files. However, changes in these files could require you to migrate to new configuration files manually. In addition, all icons and .gif files must be reapplied and references to them in PCF files redone. This is not done by the upgrade tool.

Rules – Your installation includes rules that govern its behavior at critical decision points. Your rules must reference only objects in the upgraded data model. Manual changes to rules are the only way to ensure correct references.

Gosu – Between versions, Guidewire deprecates or deletes some Gosu (formerly GScript) functions. You can not use removed functions. Manually remove deleted functions. Remove references to deprecated functions.

Integration Impacts

Your BillingCenter implementation supports many integrations with external systems. An upgrade requires that you manually update or rewrite these integrations. Guidewire provides tools and sample integrations for newer releases. Parts of this document that describe upgrading from a prior major version include a topic about upgrading integrations and Gosu from the prior major version. The *BillingCenter New and Changed Guide* and *BillingCenter Integration Guide* can also help you estimate the changes required for each integration.

Creating Project Estimates

After completing the opportunity assessment and determining the impact of product changes on your implementation, you are able to define the scope of your upgrade project. Based on that scope you can identify some options for project staffing and schedule. If required, you can also produce a cost-benefit analysis for your upgrade project based on the scope and options defined.

Preparing for the Upgrade

Depending on your organization and implementation, you might need between two to eight weeks to prepare for the upgrade. This preparation work can be performed prior to or in parallel with the upgrade assessment and project inception phases of your upgrade project. Your main task is writing an upgrade specification. Other tasks in this phase include:

- Review results of the upgrade assessment and agree upon upgrade project scope.
- Review product documentation.
- Review “Upgrading Display Languages” on page 30 in the *Globalization Guide* if upgrading with a language pack installed.
- Correct issues with database consistency checks.
- Ensure that your implementation documentation is up to date.

- Evaluate the skills and availability of internal resources.
- Identify and assign internal resources.
- Schedule and participate in training about changes and new features in the new release.
- Review and identify required changes to test scripts for the upgrade implementation.
- Acquire additional hardware and software to support infrastructure changes, if necessary.
- Set up a new environment for upgrade development.
- Set up separate branches for the upgrade in a source code control tool.

Writing an Upgrade Specification

Write an upgrade specification document that details the schedule, people, and equipment you expect to use during the upgrade. While writing the upgrade specification, answer the following questions:

- Does the upgrade to BillingCenter 8.0.4 require software or hardware that your company must purchase? What is the expected lead time for a purchase at your company?
- Is the development and test infrastructure in place to ensure the new configuration meets company standards?
- Which old configurations are you upgrading?
- Which new features do you need to configure?
- Which integrations are impacted and to what extent?
- Does your staff have the appropriate knowledge of BillingCenter, or is additional training required?
- What external support, if any, is required to execute the upgrade?
- If you are using external support, how are responsibilities divided among the team members?
- How does the company support production fixes while executing the upgrade initiative?
- If something goes wrong during the upgrade of the production environment, how do you recover?
- How does your company coordinate the production environment upgrade?
- Do you need to train your personnel on any aspects of the new system?
- Who supports the new configuration if users have questions?
- What type of documentation do you need for your new configuration?

To write an upgrade specification, take into account the testing and quality assurance your company requires. For each configuration upgrade and new feature in the configuration, define how that particular configuration will be tested and what criteria it must meet for acceptance.

Project Inception

Project inception typically takes between one to two weeks. During this phase:

- Review results of upgrade assessment..
- Finalize the project scope, resources and schedule.
- Ensure any required training has been performed by Guidewire Education.
- Make any necessary adjustments to assigned resources.
- Prepare the upgrade project plan.
- Hold workshops to review product functionality.
- Review the documented Guidewire upgrade steps and adapt them to your project.
- Perform project kick-off.

Design and Development

The design and development phase can take between two to three months depending on the extent of necessary changes. During this phase:

- Set up new environments for upgrade and performance testing.
- Define the system test plan.
- Update and creating system test scripts.
- Define user training requirements.
- Define your deployment and post go-live support plan.

System Test

Guidewire strongly recommends that you spend significant time testing your unique BillingCenter configuration. Perform testing in a development environment, not a production environment. Follow the test plan you created in the planning phase. The length of time you spend testing depends on the complexity of your installation, but typically lasts between two and three months. During this phase:

- Perform system testing, including performance and stress testing. Application hardware configurations such as clustering, load balancing, and email servers must work as expected.
- Test your entire configuration: user interface, data model and extensions, business model and the rules that implement it, and integrations to external systems.
- Perform several dry-runs of the database upgrade against a current copy of the production database. Test upgrading a copy of production data as early as possible when the upgrade includes LOB typelist changes. This provides time to address issues that might arise with production data that would not otherwise be detected.
- Perform user acceptance testing.
- Finalize training materials and deliver training.
- Document step-by-step tasks and projected schedule for deployment weekend.
- Prepare the production environment for deployment.
- Finalize plans for post go-live support.

Deployment and Support

The final step in the upgrade roadmap is to deploy the upgraded implementation into the production environment. During this stage, coordinate within your company to ensure minimum interruption to business and sufficient time to qualify the new configuration in the production environment. Guidewire recommends that you perform the deployment over a weekend, with one to two weeks allocated for post go-live support. During this phase:

- Deploy the new configuration into production.
- Upgrade the production database.
- Verify the upgrade was successful.
- Provide heightened support.

Sample Deployment Plan

This topic includes a sample deployment plan for the upgrade of ContactManager and BillingCenter.

ContactManager Prerequisite steps

1. Complete the entire configuration upgrade (code merge) and all functional and non-functional testing against an upgraded copy of production.
2. During the configuration upgrade, preserve MatchSetKey column data if you want to keep it. This only applies to upgrades from 6.0 or from 7.0 versions prior to 7.0.6.
3. Confirm that ContactManager 8.0.4 Studio shows no compilation errors, and ideally no warnings.
4. Confirm that all components of the upgraded infrastructure will be on a supported release, per the Platform Support Matrix.
5. In production, validate the database schema using the `system_tools -verifydbschema`. Ideally, there will be no issues reported.
6. In production, check the database consistency from the **Server Tools** → **Info Pages** → **Consistency Checks** page. Resolve any errors that are reported.
7. In production, generate a data distribution report. Confirm that the report can be generated properly.
8. In production, if database statistics are not already current, update database statistics shortly before the upgrade deployment.
9. For SQL Server: Decide whether to enable migration to 64-bit IDs during or after the upgrade.
10. For SQL Server: Confirm that the collation setting in the database matches the setting defined in `collations.xml` in the upgraded code base.
11. Prepare all infrastructure, including the database and application servers, load balancer, and so forth.
12. Preserve upgrade instrumentation from the most recent upgrade, if you want to keep it. This information can be downloaded from the **Upgrade Info** page.

Contact Manager Deployment Steps

Perform steps 1-11 in the current production environment.

1. Confirm all batch processing has completed. Ensure no batch process or work queue has a **Status of Active** on the **Server Tools** > **Batch Process Info** page.
2. Suspend all message destinations from the **Administration** > **Event Messages** page.
3. Purge completed inactive messages. Note: You can also purge some in advance of the deployment window.
4. Purge completed workflows. Note: You can also purge some in advance of deployment window.
5. Purge completed workflow logs. Note: You can also purge some in advance of deployment window.
6. Validate the database schema using the `system_tools -verifydbschema` command. Ideally, there will be no issues reported. Drop unused columns if necessary, either before or after the upgrade.
7. Check the database consistency from the **Server Tools** → **Info Pages** → **Consistency Checks** page. Ensure no new errors are reported.
8. Generate a data distribution report.
9. Shut down production application servers.
10. Create a database backup.
11. Confirm adequate disk space for the database during upgrade. Allot at least 150% of the current production database size.
12. Disable database replication.

13. For Oracle: Assign default tablespace.
14. For Oracle: Optionally disable logging, statistics update, and statistics update for tables with locked statistics.
15. For SQL Server: Optionally disable SQL Server logging.
16. Move the database from old RDBMS release to new RDBMS release. This step applies to major version upgrades only.
17. For SQL Server: For major version upgrades, set the compatibility level to 110 with the command:

```
ALTER DATABASE dbname SET COMPATIBILITY_LEVEL = 110
```
18. Upgrade application servers or repoint the production URL to new application servers as appropriate. This step applies to major version upgrades only.
19. Disable the scheduler in `config.xml`.
20. Enable the database upgrade in `database-config.xml`.
21. Deploy WAR with upgraded configuration to application servers.
22. Start the server to begin database upgrade.
23. Review server log for unexpected errors. Search for the string `ERROR` in the log.
24. Perform initial high-level, view-only testing to ensure system availability.
25. Validate the database schema using the `system_tools -verifydbschema` command. Ideally, there will be no issues reported.
26. Check the database consistency from the `Server Tools → Info Pages → Consistency Checks` page. Compare the results with the results from the prerequisite steps.
27. Generate a data distribution report. Compare it against the data distribution report generated during step 8.
28. Execute custom data validation scripts.
29. Run the Deferred Upgrade Tasks batch process if needed.
30. Run Phone Number Normalizer. Run Phone Number Normalizer on ContactManager first. This step applies to major version upgrades only.
31. Stop the server.
32. Create a database backup.
33. Enable the scheduler in `config.xml`.
34. Disable the database upgrade in `database-config.xml`.
35. Deploy new WAR.
36. Start the server. The upgrade is now completed.
37. Update database statistics by generating the statistics updating statements and executing them.
38. Perform initial testing of key application functionality.
39. Send announcement to user community.
40. For SQL Server: Migrate to 64-bit IDs if not done as part of initial upgrade.

BillingCenter Prerequisite steps

1. Complete the entire configuration upgrade (code merge) and all functional and non-functional testing against an upgraded copy of production.
2. Confirm that BillingCenter 8.0.4 Studio shows no compilation errors, and ideally no warnings.

3. Confirm that all components of the upgraded infrastructure will be on a supported release, per the Platform Support Matrix.
4. Confirm that user workstations will have a supported browser and, if appropriate, be able to work with documents.
5. In production, validate the database schema using the `system_tools -verifydbschema`. Ideally, there will be no issues reported.
6. In production, check the database consistency from the **Server Tools** → **Info Pages** → **Consistency Checks** page. Resolve any errors that are reported.
7. In production, generate a data distribution report. Confirm that the report can be generated properly.
8. In production, if database statistics are not already current, update database statistics shortly before the upgrade deployment.
9. For SQL Server: Decide whether to enable migration to 64-bit IDs during or after the upgrade.
10. For SQL Server: Confirm that the collation setting in the database matches the setting defined in `collations.xml` in the upgraded code base.
11. Prepare all infrastructure, including the database and application servers, load balancer, and so forth.
12. Preserve upgrade instrumentation from the most recent upgrade, if you want to keep it. This information can be downloaded from the **Upgrade Info** page.

BillingCenter Deployment Steps

Perform steps 1-11 in the current production environment.

1. Run Commission Payable Calculations batch processing if applicable.
2. Confirm all batch processing has completed. Ensure no batch process or work queue has a **Status of Active** on the **Server Tools** > **Batch Process Info** page.
3. Suspend all message destinations from the **Administration** > **Event Messages** page.
4. Purge completed inactive messages. Note: You can also purge some in advance of the deployment window.
5. Purge completed workflows. Note: You can also purge some in advance of deployment window.
6. Purge completed workflow logs. Note: You can also purge some in advance of deployment window.
7. Validate the database schema using the `system_tools -verifydbschema` command. Ideally, there will be no issues reported. Drop unused columns if necessary, either before or after the upgrade.
8. Check the database consistency from the **Server Tools** → **Info Pages** → **Consistency Checks** page. Ensure no new errors are reported.
9. Generate a data distribution report.
10. Shut down production application servers.
11. Create a database backup.
12. Confirm adequate disk space for the database during upgrade. Allot at least 150% of the current production database size.
13. Disable database replication.
14. For Oracle: Assign default tablespace.
15. For Oracle: Optionally disable logging, statistics update, and statistics update for tables with locked statistics.
16. For SQL Server: Optionally disable SQL Server logging.

17. Move the database from old RDBMS release to new RDBMS release. This step applies to major version upgrades only.
18. For SQL Server: For major version upgrades, set the compatibility level to 110 with the command:

```
ALTER DATABASE dbname SET COMPATIBILITY_LEVEL = 110
```
19. Upgrade application servers or repoint the production URL to new application servers as appropriate. This step applies to major version upgrades only.
20. Disable the scheduler in config.xml.
21. Enable the database upgrade in database-config.xml.
22. Deploy WAR with upgraded configuration to application servers.
23. Start the server to begin database upgrade.
24. Review server log for unexpected errors. Search for the string ERROR in the log.
25. Perform initial high-level, view-only testing to ensure system availability.
26. If using Guidewire rating data, reload rating sample data.
27. Validate the database schema using the `system_tools -verifydbschema` command. Ideally, there will be no issues reported.
28. Check the database consistency from the **Server Tools** → **Info Pages** → **Consistency Checks** page. Compare the results with the results from the prerequisite steps.
29. Generate a data distribution report. Compare it against the data distribution report generated in step 9.
30. Execute custom data validation scripts.
31. Run the Deferred Upgrade Tasks batch process if needed. Launch this process from the `maintenance_tools`.
32. Run Phone Number Normalizer. Run Phone Number Normalizer on ContactManager first. This step applies to major version upgrades only.
33. Stop the server.
34. Create a database backup.
35. Enable the scheduler in config.xml.
36. Disable the database upgrade in database-config.xml.
37. Deploy new WAR.
38. Start the server. The upgrade is now completed.
39. Update database statistics by generating the statistics updating statements and executing them.
40. Perform initial testing of key application functionality.
41. Send announcement to user community.
42. For SQL Server: Migrate to 64-bit IDs if not done as part of initial upgrade.
- 43.

Upgrading from 8.0.x

This part describes how to perform an upgrade from BillingCenter 8.0.x to 8.0.4.

If you are upgrading from BillingCenter 7.0.x, see “Upgrading from 7.0.x” on page 93 instead.

If you are upgrading from BillingCenter 3.0.x, see “Upgrading from 3.0.x” on page 201 instead.

This part includes the following topics:

- “Upgrading the BillingCenter 8.0.x Configuration” on page 29
- “Upgrading the BillingCenter 8.0.x Database” on page 51
- “Upgrading BillingCenter from 8.0.x for ContactManager” on page 87
- “Upgrading ContactManager from 8.0.x” on page 89

Upgrading the BillingCenter 8.0.x Configuration

This topic describes how to upgrade the BillingCenter configuration from version 8.0.x to 8.0.4.

If you are upgrading from a 7.0.x version, see “Upgrading the BillingCenter 7.0.x Configuration” on page 95 instead.

If you are upgrading from a 3.0.x version, see “Upgrading the BillingCenter 3.0.x Configuration” on page 203 instead.

This topic includes:

- “Overview of ContactManager Upgrade” on page 30
- “Obtaining Configurations and Tools” on page 30
- “Creating a Configuration Backup” on page 34
- “Removing Patches” on page 34
- “Removing Language Packs” on page 34
- “Updating Infrastructure” on page 34
- “Launching the BillingCenter 8.0.4 Configuration Upgrade Tool” on page 35
- “Configuration Upgrade Tool Automated Steps” on page 36
- “Using the BillingCenter 8.0.4 Upgrade Tool Interface” on page 36
- “Configuration Merging Guidelines” on page 42
- “Data Model Merging Guidelines” on page 43
- “Merging Display Properties” on page 45
- “Upgrading Rules to BillingCenter 8.0.4” on page 46
- “Translating New Display Properties and Typecodes” on page 47
- “Validating the BillingCenter 8.0.4 Configuration” on page 47
- “Building and Deploying BillingCenter 8.0.4” on page 48

Overview of ContactManager Upgrade

The automatic upgrade process for ContactManager is almost precisely the same as for BillingCenter. However, there are differences, especially for manual upgrade. Additionally, Guidewire recommends that you complete the ContactManager upgrade before manually updating files that BillingCenter uses for integration with ContactManager.

- For information on upgrading ContactManager, see “Upgrading ContactManager from 8.0.x” on page 89.
- For information on upgrading BillingCenter files used to integrate with ContactManager, see “Upgrading BillingCenter from 8.0.x for ContactManager” on page 87.

Obtaining Configurations and Tools

Configuration refers to everything related to the application except the database. This includes configuration files such as typelists and PCF files, the file structure, web resources, Gosu classes, rules, plugins, libraries, localization files, and application server files.

The upgrade process involves three configurations. This guide defines and refers to these configurations as base, customer, and target.

Base – The unedited, original configuration on which you based your customer configuration. The base configuration is included in directories within `/modules` other than `/configuration`.

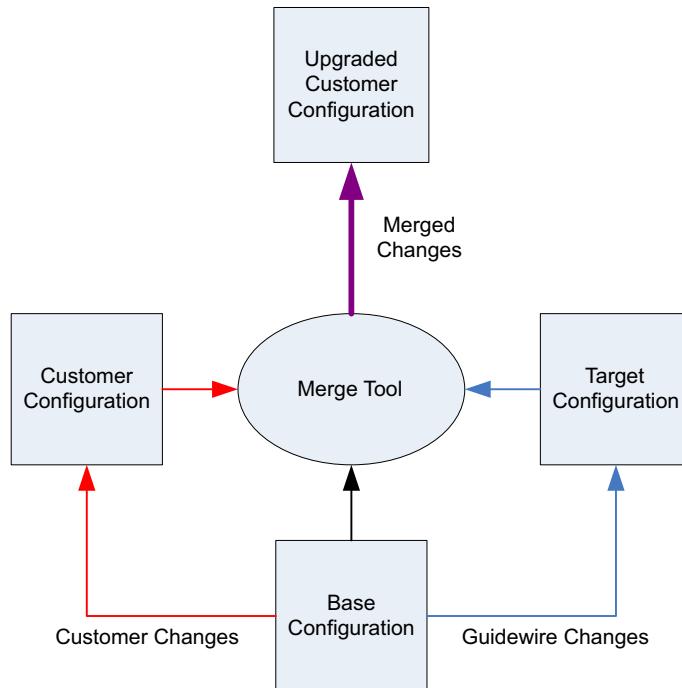
Customer – The configuration you are now using and will upgrade. This is the base configuration of the BillingCenter version that you currently run with your custom configuration applied. Custom configuration files are stored in the `modules/configuration` directory. The Configuration Upgrade Tool compares your customized versions of files in the `modules/configuration` directory and its subdirectories with files in the target BillingCenter version. If you have customized files outside of `modules/configuration` and its subdirectories, manually upgrade those files by comparing them with versions of those files in the target BillingCenter version.

Target – The unedited, original configuration of BillingCenter 8.0.4 on which your upgraded configuration will be based. Guidewire grants you access to the Guidewire Resource Portal, from which you download the target configuration ZIP file. Unzip the target BillingCenter 8.0.4 configuration into another directory. Download the latest patch release for the target version that you are downloading. Follow the instructions with the patch release to install it after you unzip the target version. Do not make any modifications to the target configuration prior to completing the configuration upgrade. Do not start Guidewire Studio for the target configuration until you have completed the configuration upgrade.

IMPORTANT Set all files in the base, customer, and target configurations to writable before beginning the upgrade.

The following figure shows how you use these configurations to create a merged configuration. The merged configuration combines your changes to the original base configuration (the customer configuration) and Guidewire

changes to the base configuration (the target configuration). The original base configuration provides a basis for comparison.



Viewing Differences Between Base and Target Releases

To view an inventory of the differences between the base release and the target release, download and carefully review the *Upgrade Diffs Report* from the Guidewire Resource Portal.

1. Open a browser to <https://guidewire.hivelive.com/pages/home>.
2. Click Project Center → Upgrade Services.
3. Click Review the Upgrade Diff Reports.
4. Click BillingCenter.
5. Click Upgrade Diff Reports - BillingCenter or Upgrade Diff Reports - ContactManager.
6. Click Upgrade From *base version*.
7. Click Upgrade To 8.0.4.

Specifying Configuration and Tool Locations

The BillingCenter 8.0.4 Configuration Upgrade Tool depends on the following tools:

- **Text Editor** – An ASCII text editor you use to edit programs and similar files, for example, Notepad, WordPad or Textpad. This editor must not put additional characters in files, as Word does.

- **Merge Tool** – An editor which displays two or three versions of a file, highlights the differences between them, and allows you to perform basic editing functions on them. Also known as a “diff tool.” Examples include Araxis Merge Professional and Beyond Compare Professional. If using Beyond Compare Professional, see “Considerations for Using Beyond Compare Professional” on page 33. Configure the merge tool to ignore end of line characters to reduce the number of potential false positives during the configuration upgrade step.

IMPORTANT The merge tool that you use must support three-way file comparison and merging. During the configuration upgrade, for some files you will need to compare three versions: the original base version, the new version and your customized version.

The Configuration Upgrade Tool needs the location of the BillingCenter environment that you will upgrade. The tool stores all versions of files to be merged and merge results in a `tmp` directory that it creates within the target environment. Define paths to the configuration and tools in the `BillingCenter/modules/ant/upgrade.properties` file of the target BillingCenter 8.0.4 environment. Remove the pound sign, ‘#’, preceding each property to uncomment the line and then specify values. Use double backslashes in paths. For example, `C:\\\\BillingCenter`. You do not need to use quotes for paths that include spaces.

The following properties are configurable in `upgrade.properties`.

Property	Description
<code>upgrader.priorversion.dir</code>	Path to the top-level BillingCenter directory of the current customer environment. This directory contains <code>/bin</code> and other BillingCenter directories.
<code>upgrader.editor.tool</code>	Path to an executable editing tool.
<code>upgrader.diff.tool</code>	Path to an executable difference editor tool, such as Araxis Merge Professional or Beyond Compare Professional, also known as a merge tool, used for two-way merges. If your difference editor supports both two and three-way merges, you can use the same value for <code>upgrader.diff.tool</code> and <code>upgrader.merge.tool</code> .
<code>upgrader.merge.tool</code>	Path to an executable difference editor tool, such as Araxis Merge Professional or Beyond Compare Professional, also known as a merge tool, used for three-way merges. The merge tool specified for <code>upgrader.merge.tool</code> must support three-way file comparison and merging. If your difference editor supports both two and three-way merges, you can use the same value for <code>upgrader.diff.tool</code> and <code>upgrader.merge.tool</code> . You might need to configure the display of your merge tool to show three panels.

Property	Description
upgrader.merge.tool.arg.order	<p>The order of command line arguments to the difference editor tool specified by <code>upgrader.merge.tool</code>.</p>
	<p>The command line arguments available to the difference editor typically include the display order, from left to right, for versions of a file viewed in the difference editor.</p>
	<p>The available options are:</p>
	<p><code>NewBase</code> is the unmodified target version provided with BillingCenter 8.0.4.</p>
	<p><code>PriorBase</code> is the original base version.</p>
	<p><code>PriorCustom</code> is your configured version.</p>
	<p><code>Resulting</code> is the merged output file.</p>
	<p>The order of these values controls the display order in the difference editor tool. If the tool displays just two versions, it uses the same relative order.</p>
	<p>By default, the display order places the old base version of a file in the center. The old base version is the common ground between the new uncustomized version and the old customized version. Guidewire changed the old base version to create the new target version, and you changed the old base version to create the customized version in your configuration. With the old base version in the center, you can incorporate both sets of changes to create a customized target version.</p>
	<p>The default order enables you to merge changes from the left and right to the center and save the merged version. If you use another difference editor, you might need a different order to achieve the same result. Samples are shown below for various difference engines:</p>
	<p>Araxis Merge Professional</p>
	<pre>upgrader.merge.tool.arg.order = NewBase PriorBase PriorCustom</pre>
	<p>Beyond Compare Professional</p>
	<pre>upgrader.merge.tool.arg.order = NewBase PriorCustom PriorBase Resulting</pre>
	<p>P4Merge</p>
	<pre>upgrader.merge.tool.arg.order = PriorBase NewBase PriorCustom</pre>
	<p>You might need to configure the display of your merge tool to show three panels.</p>
upgrader.steps.class	<p>The class to run to execute the configuration upgrade automated steps. If you are upgrading BillingCenter 3.0 or newer, then leave this property commented out.</p>
exclude.pattern	<p>A regular expression pattern for paths of files for the Configuration Upgrade Tool to mark as unmergeable. Typically, you use <code>exclude.pattern</code> to specify source control metadata files. Samples are provided in <code>upgrade.properties</code> for CVS and SVN.</p>

Considerations for Using Beyond Compare Professional

By including the `Resulting` parameter on the `upgrader.merge.tool.arg.order` property for Beyond Compare, you can avoid an extra manual step to specify the output file on each merge.

To configure the Configuration Upgrade Tool for Beyond Compare Professional

1. Create a new file in a text editor.
2. Add the following text to the new file, modifying the location of the Beyond Compare Professional executable if necessary.


```
@echo off
"C:\Program Files (x86)\Beyond Compare 3\BCompare.exe" %1 %2 %3 %4
```
3. Save the file as `merge.cmd` in the `BillingCenter/modules/ant` directory.
4. Open the `BillingCenter/modules/ant/upgrade.properties` file of the target BillingCenter 8.0.4 environment.

5. Set the `upgrader.diff.tool` property to the location of the Beyond Compare Professional executable. For example:
`upgrader.diff.tool = "C:\\Program Files (x86)\\Beyond Compare 3\\BCompare.exe"`
6. Set the `upgrader.merge.tool` property to a command line script file. For example:
`upgrader.merge.tool = ".\\modules\\ant\\merge.cmd"`
7. Set the `upgrader.merge.tool.arg.order` property:
`upgrader.merge.tool.arg.order = NewBase PriorCustom PriorBase Resulting`
8. Set the remaining properties in `upgrade.properties` as described in the table.
9. Save `upgrade.properties`.

Creating a Configuration Backup

Prepare the environment so that you can make a total recovery of the original installation if you run into problems during the upgrade.

Guidewire recommends that you track BillingCenter configuration changes in a source code control system. Before upgrading, have a labeled version of your entire pre-upgrade BillingCenter configuration folder (`modules/configuration`) from your custom configuration. A labeled version is a named collection of file revisions.

As an even stronger precaution, make a backup of the same installation directories.

Removing Patches

If you have applied any patches from Guidewire to BillingCenter, remove the patches before you run the configuration upgrade. Patches are specific to the pre-upgrade version. If you do not remove the JAR files for patches, the Configuration Upgrade Tool copies the JAR files to the upgraded configuration. In that case, your upgraded configuration will be using a JAR file that is not current. Patches are typically installed by adding JAR files within `modules/configuration/deploy`.

Removing Language Packs

If you have language packs installed, you must remove the language packs before upgrading BillingCenter. See “Upgrading Display Languages” on page 30 in the *Globalization Guide*.

Updating Infrastructure

Before starting the upgrade, have the supported server operating systems, application server and database software, JDK, and client operating systems for the target version. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

For SQL Server, after you upgrade the database server software, run the following command to set the compatibility level:

```
ALTER DATABASE databaseName SET COMPATIBILITY_LEVEL = 110
```

Launching the BillingCenter 8.0.4 Configuration Upgrade Tool

To upgrade your configuration, merge Guidewire changes to the base configuration with your changes. The *Configuration Upgrade Tool*, provided by Guidewire with the target BillingCenter 8.0 configuration, facilitates this process. The tool compares your customized versions of files in the `modules/configuration` directory and its subdirectories with files in the target BillingCenter version. If you have customized files outside of `modules/configuration` and its subdirectories, manually upgrade those files by comparing them with versions of those files in the target BillingCenter version.

The Configuration Upgrade Tool requires two tools: a merge tool such as Araxis Merge Professional or Beyond Compare 3 Professional, and a text editor. Configure the merge tool to ignore end-of-line characters to reduce the number of potential false positives during the configuration upgrade step.

IMPORTANT The merge tool that you use must support three-way file comparison and merging. During the configuration upgrade, for some files you will need to compare three versions: the original base version, the new version and your customized version.

The Configuration Upgrade Tool performs a series of automated steps and then opens an interface that you use for the manual merge process.

Guidewire can provide guidance on using the Configuration Upgrade Tool in a multi-user environment using a source control management system.

See the *Upgrade Diffs Report* for an inventory of the differences between the base release and the target release. To retrieve the *Upgrade Diffs Report* follow the procedure described in “Viewing Differences Between Base and Target Releases” on page 97.

Also see the *BillingCenter New and Changed Guide* for a description of new features and changes to existing features. Review key data model changes as these changes might impact customizations in your system.

To launch the Configuration Upgrade Tool

1. Open a command window.
2. Navigate to the `modules/ant` directory of the target configuration.
3. Execute the following command:

```
ant -f upgrade.xml upgrade > upgrade_log.txt
```

You can specify any file to log messages and exceptions.

The Configuration Upgrade Tool first copies the modules of the base environment to a `tmp/cfg-upgrade/modules` directory in the target environment. The base environment is specified by the `upgrader.priorversion.dir` property in `modules/ant/upgrade.properties` in the target environment.

The Configuration Upgrade Tool then performs a number of automated steps, described later in this topic. Once the tool completes the automated steps, it opens a user interface. The interface opens whether the automated steps were successful or not. Review the log file or console before proceeding with the manual merge process.

Restarting the Configuration Upgrade Tool

The Configuration Upgrade Tool stores work in progress by recording which files you have marked resolved in the `accepted_files.lst` file. This file is stored in the `merge` directory of the target environment. You can close the interface and restart it later without losing your work in progress.

If you do want to start the upgrade over, use the `clean` command to empty the working directories.

```
ant -f upgrade.xml clean
```

WARNING If you empty the `tmp` directory after beginning to merge, you lose all completed merges that you have not resolved and moved into the target configuration directory.

Configuration Upgrade Tool Automated Steps

The Configuration Upgrade Tool prepares for the manual configuration merge process by performing a number of automated steps. Review these steps before proceeding with the configuration merge. Understanding these automated steps helps to understand some file changes you will see when merging the configuration. Finally, some steps might require manual intervention if there is an issue.

Copying Custom Rules and Adding BillingCenter 8.0.4 Default Rules

The upgrade copies customized rules to the target configuration `modules/configuration/config/rules` directory.

This step also copies the default rules provided with BillingCenter 8.0.4 to a BillingCenter 8.0.4 folder within the `modules/configuration/config/rules` directory of the target configuration. This is so you have a copy of the default rules in a folder in Studio that you can use to compare with your custom rules.

Using the BillingCenter 8.0.4 Upgrade Tool Interface

IMPORTANT Review the automated step descriptions before you proceed. Some automated steps might require you to perform a manual step while merging the configuration. Typically, such automated steps insert a warning into a file. Check the `steps_results.txt` file for warning and error messages. Correct any issues reported. Then, delete `steps_results.txt` and restart the Configuration Upgrade Tool.

After the Configuration Upgrade Tool completes the automated steps, the working area contains up to three versions of the same file:

- The *customer* file.
- The *base* file, from which you configured the customer file.
- The *target* file, from BillingCenter 8.0.4.

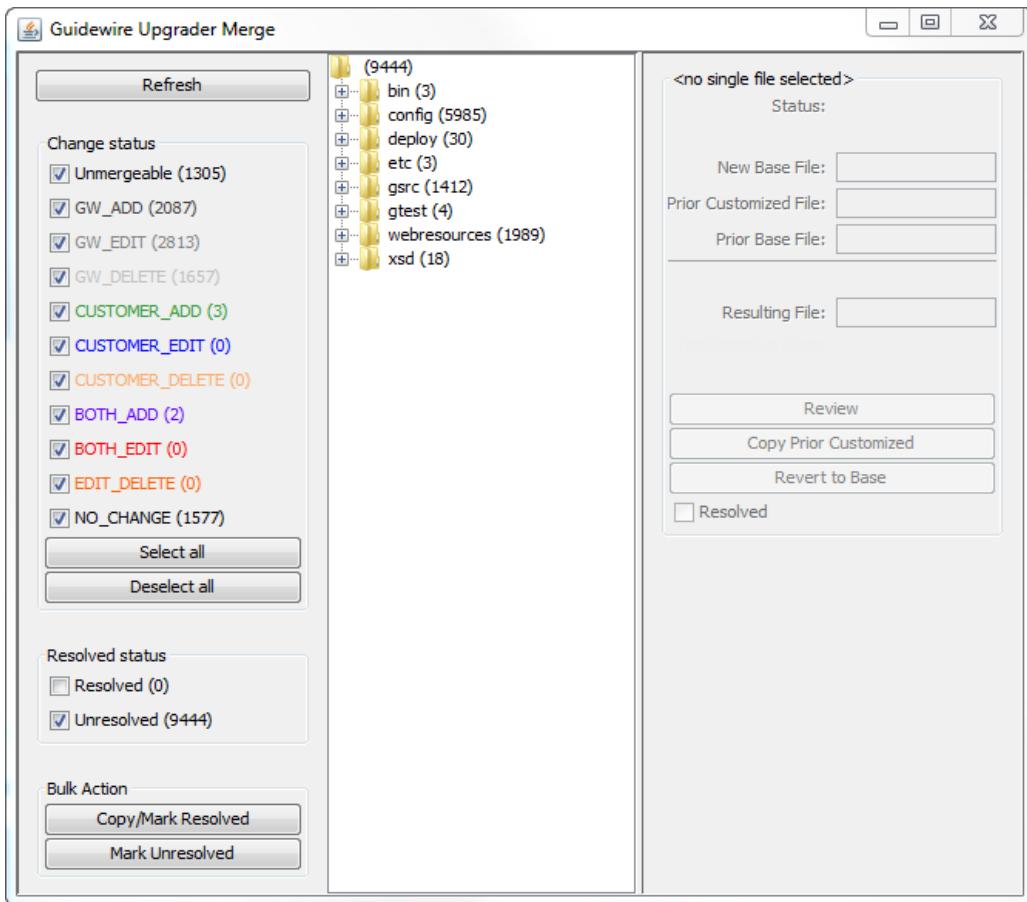
In the manual process of the upgrade, you decide whether to use one of these versions unchanged, or merge versions together. The Configuration Upgrade Tool provides a user interface to assist with the manual process. This interface has several important functions:

- It shows a complete list of all configuration files.
- It allows you to filter this list. You can, for example, view a list of all files that differ between the target version and your version. See “Change Status Filters” on page 38.
- It displays two or three versions of a file and their differences, using a merge tool you supply, such as Araxis Merge or P4Merge, defined in `upgrade.properties`.
- It lets you edit your file, incorporating changes from the other file versions, and save it.
- It lets you accept this merged version instead of one of the previous versions.

- It lets you edit the file after you have accepted changes from the merge using the text editor defined in `upgrade.properties`.

After you have accepted or merged all files that the Configuration Upgrade Tool displays, the merging process is complete.

The Configuration Upgrade Tool displays three panels. The center panel is a tree view of the files in the configuration, filtered by filter choices selected in the left panel. Files appear in the color of the filter that found them. As you select a file in the center panel, the right panel displays file information and buttons to perform actions on that file.



Filters

The left panel of the Configuration Upgrade Tool contains:

- Refresh Button
- Change Status Filters
- Resolved Status Checkboxes
- Bulk Action Buttons

Refresh Button

If multiple users are working in the same directory, each user can mark files as resolved. The Refresh button refreshes the resolved status of files shown in the Configuration Upgrade Tool for changes contributed by all users working in the same directory.

Change Status Filters

This table lists the change status filters that the Configuration Upgrade Tool displays in the left panel. Use the check boxes next to the filters to select one or any combination of change statuses to view. Use the **Select all** or **Deselect all** buttons to select or deselect all filters. The following table describes change status filters. The Guidewire Action column lists the change Guidewire has made to files matching a status filter since the prior version. The Your Action column lists the change to the file in your implementation:

Merge Status	Guidewire Action	Your Action	Type of change made to file	Action taken by Configuration Upgrade Tool
Unmergeable	change format of file	any	file exists in a different format and thus cannot be merged with an old version	If you resolve the file, the Configuration Upgrade Tool takes no action. The file, in the new format, already exists in the target configuration. The Configuration Upgrade Tool automatically marks certain files as unmergeable, including rules files. The Configuration Upgrade Tool upgrades these files before the interface displays. You can also specify a regular expression pattern in upgrade.properties for file paths to mark files matching that pattern as unmergeable. Set the pattern as the value of the exclude.pattern property.
GW_ADD	add	none	file in target not in base	If you resolve the file, the Configuration Upgrade Tool takes no action. The file added by Guidewire already exists in the target configuration. Double-clicking opens the file in the text editor specified by upgrader.editor.tool in upgrade.properties.. If you make changes, the tool prompts you to copy the file to the target configuration.
GW_EDIT	edit	none	file in target differs from base	If you resolve the file, the Configuration Upgrade Tool takes no action. The file added by Guidewire already exists in the target configuration. Double-clicking opens the file in the merge tool specified by upgrader.diff.tool in upgrade.properties to perform a comparison between the new Guidewire version and the original base version. If you make changes, the tool prompts you to copy the file to the target configuration.
GW_DELETE	delete	none	file in base not in target	If you resolve the file, the Configuration Upgrade Tool takes no action. The file deleted by Guidewire no longer exists in the target configuration. Double-clicking opens the file in the text editor specified by upgrader.editor.tool in upgrade.properties.. If you make changes, the tool prompts you to copy the file to the target configuration.
CUSTOMER_ADD	none	add	file in customer configuration only	If you resolve the file, the Configuration Upgrade Tool copies the file to the target configuration if the file has not been copied there already. Double-clicking opens the file in the text editor specified by upgrader.editor.tool in upgrade.properties.. If you make changes, the tool prompts you to copy the file to the target configuration.

Merge Status	Guidewire Action	Your Action	Type of change made to file	Action taken by Configuration Upgrade Tool
CUSTOMER_EDIT	none	edit	file differs between customer and base configurations	If you resolve the file, the Configuration Upgrade Tool copies the file to the target configuration if the file has not been copied there already.
			file unchanged between base and target configurations	Double-clicking opens the file in the merge tool specified by <code>upgrader.diff.tool</code> in <code>upgrade.properties</code> to perform a comparison between your custom version and the original base version. If you make changes, the tool prompts you to copy the file to the target configuration.
CUSTOMER_DELETE	none	delete	file exists in the base and target configurations but not in the customer configuration	If you click Delete , the Configuration Upgrade Tool removes the file from the target configuration. If you click Revert to Base , the Configuration Upgrade Tool leaves the file in the target configuration.
BOTH_ADD	add	add	new file with matching name in both target and customer configurations (rare)	You must either merge the two versions of the file or copy your prior version of the file into the target configuration before you can resolve the file. Double-clicking opens the file in the merge tool specified by <code>upgrader.diff.tool</code> in <code>upgrade.properties</code> to perform a merge between your version and the Guidewire version. If you make changes, the tool prompts you to copy the merged file to the target configuration.
BOTH_EDIT	edit	edit	file changed in both customer and target configurations	You must either merge the two versions of the file or copy your prior version of the file into the target configuration before you can resolve the file. Double-clicking opens the file in the merge tool specified by <code>upgrader.merge.tool</code> in <code>upgrade.properties</code> to perform a three-way merge between your custom version and the updated Guidewire version. If you make changes, the tool prompts you to copy the merged file to the target configuration.

Merge Status	Guidewire Action	Your Action	Type of change made to file	Action taken by Configuration Upgrade Tool
EDIT_DELETE	delete	edit	file changed from base in customer configuration and does not exist in target configuration	<p>If you resolve the file, the Configuration Upgrade Tool takes no action.</p> <p>Double-clicking the file opens your customized file and the original base file in the merge tool specified by <code>upgrader.diff.tool</code> in <code>upgrade.properties</code>. When you close the merge tool, the Configuration Upgrade Tool prompts you to copy the file to the target configuration. If you are sure you want your customized version, you can click Copy prior customized to move the file to the target configuration.</p> <p>The EDIT_DELETE flag appears on a file when your configuration has a customized version of the file but Guidewire has deleted the file from that location. There are two possible reasons for this deletion. One reason is that Guidewire removed the file from BillingCenter. The second reason is that Guidewire has moved the file to a different folder.</p> <p>If Guidewire has completely removed the file, review the <i>BillingCenter New and Changed Guide</i>, release notes, and the Upgrade Diff report for descriptions of the change affecting the deleted file. Then determine if you want to continue moving your customization to the new or changed feature. If not, then the customization will be lost.</p> <p>For the second scenario, find where the file has been moved by searching the target version. Move your customized file to the same location in the working directory and make sure to match any case changes in the filename. When you refresh the list of merge files, the file now appears under the CUSTOMER_EDIT filter. You can now proceed with the merge. If you do not move the file over, you can instead perform the merge manually by opening both files and incorporating the changes.</p>
NO_CHANGE	none	none	file not changed from base configuration in either customer or target configurations	<p>If you resolve the file, the Configuration Upgrade Tool takes no action. The file already exists in the target configuration.</p> <p>Double-clicking opens the file in the text editor specified by <code>upgrader.editor.tool</code> in <code>upgrade.properties</code>. If you make changes, the tool prompts you to copy the file to the target configuration.</p>

Resolved Status Checkboxes

Beneath the change status filters are checkboxes to toggle the visibility of resolved and unresolved files. Use these checkboxes with the change status filters to specify which types of files you want visible in the center panel. For example, you could select **BOTH_EDIT** and **Unresolved** to see files edited in your configuration that have also been updated by Guidewire and are not yet resolved.

The purpose of the resolved status is to have a general idea of the progress you are making in the upgrade. The tool shows the resolved status of the current file (right panel) and the total number of resolved and unresolved files (left panel).

A resolved file is simply a file that you have marked resolved. It does not relate to whether file merging or accepting has occurred.

Bulk Action Buttons

The following buttons in The **Bulk Action** part of the left panel enable you to change the resolved status of a group of selected files:

- **Copy/Mark Resolved**
- **Mark Unresolved**

You can select either one or several files and directories before using these buttons. Use the CTRL key to select multiple files and directories. Selecting a directory selects all files within that directory. You can select all files that match the filters you set by selecting the top-level directory.

After you click **Copy/Mark Resolved**, the Configuration Upgrade Tool opens a dialog detailing the actions it is about to perform.

The tool copies files matching the **CUSTOMER_ADD** and **CUSTOMER_EDIT** filters to the target configuration. If there is already a version of a file in the target configuration, then the tool does not copy the file. A file would be there already if you edited the file and clicked Yes when the tool prompted you to copy the file to the target configuration.

The tool does not do any copying for files matching the **GW_ADD**, **GW_DELETE**, **GW_EDIT**, **NO_CHANGE**, or **Unmergeable** filters. Files matching **GW_ADD**, **GW_EDIT**, **NO_CHANGE**, or **Unmergeable** filters are already present in the target version. Files matching the **GW_DELETE** filter are not in BillingCenter 8.0.4.

You can not bulk resolve multiple files that match the **BOTH_ADD**, **BOTH_EDIT**, or **EDIT_DELETE** filters. Files matching these filters require individual attention. Use the right panel of the Configuration Upgrade Tool to control merging, copying and resolving of these files.

Configuration File Tree

The center panel displays the configuration file tree. Files are color-coded to match filter colors. Files are shown one time, regardless of the number of configurations in which they exist. For information on which configurations a file exists in, select the file and view the right panel. The number of files in each directory that match the selected change status and resolved status filters is shown in parentheses.

File Details Panel

The right panel displays file details for the file you are currently examining, including:

- **Status** – The change status of the file. See “Change Status Filters” on page 38.
- **New base file** – The new version of this file supplied by Guidewire with BillingCenter 8.0.4. If there is not a Guidewire version of this file, such as for **CUSTOMER_ADD**, **EDIT_DELETE** or **GW_DELETE** files, this field is blank.
- **Prior customized file** – The locally customized version of this file from the prior version. If there is not a customized version of this file, such as for **GW_ADD**, **GW_DELETE**, **GW_EDIT** or **NO_CHANGE** files, this field is blank.
- **Prior base file** – The base version of this file in the working directory. If there is not a Guidewire version of this file in the prior base version you are upgrading from, such as for **CUSTOMER_ADD** files, this field is blank.
- **New customized file** – The customized version of this file in the BillingCenter 8.0.4 configuration directory.

The right panel fields are blank if you have multiple files selected.

File Details Panel Actions

The following buttons appear below the file details display in the right panel after you have selected a file:

- **View** – Opens the file in the editor specified in `upgrade.properties`. This button appears for files that are not customized and do not require merging, matching **GW_ADD**, **GW_EDIT**, or **GW_DELETE** filters. Only one of the **View**, **Edit** or **Merge** buttons displays, depending on the file change status.

- **Edit** – Opens the file in the editor specified in `upgrade.properties`. This button appears for custom files that do not require merging, matching the `CUSTOMER_ADD` or `EDIT_DELETE` filters.
- **Merge** – Opens the different versions of the file in the merge tool specified in `upgrade.properties`. This button appears for files that require merging, matching the `BOTH_ADD` or `BOTH_EDIT` filters.
- **Copy prior customized** – Copies the prior customized version of the file to the target configuration. This button is enabled if there is a prior customized version of the file. So it is enabled for files matching `CUSTOMER_ADD`, `CUSTOMER_EDIT`, `BOTH_ADD`, `BOTH_EDIT`, or `EDIT_DELETE` filters.
- **Delete new customized** – Remove the customized version from the target configuration. This reverses the **Copy prior customized** button action. This button is disabled until you have copied a prior customized version of the file into the target configuration.
- **Resolved** – Check this box to label the file resolved. Use the **Resolved** checkbox in the right pane to change the status of a single file. Selecting the **Resolved** checkbox does not copy the file. Use the buttons above this checkbox to handle copying or merging of file versions. You must first unresolve a file before either using the **Delete new customized** action or reapplying changes or merges.

Accepting Files that Do Not Require Merging

The following filters show lists of files that normally do not require merging.

- `CUSTOMER_ADD`
- `CUSTOMER_EDIT`
- `GW_ADD`
- `GW_EDIT`
- `NO_CHANGE`
- `Unmergeable`

You can click the **Copy/Mark Resolved** button in the left panel to resolve groups of these files.

Merging and Accepting Files

Files matching the `BOTH_ADD` and `BOTH_EDIT` filters must be merged before being accepted. Your version must be reconciled with the Guidewire target or base version. In some cases, even if only a single version of the file exists, you might want to look at it before accepting it.

You can use the `pcf.xsd` file in the `modules` directory of the target version to validate merged PCF files.

After you are satisfied with any changes, save the file. This saves the file in a temporary directory. When you close the editor or merge tool, the Configuration Upgrade Tool asks if you want to copy the file to the target configuration. If you click **Yes** (or press `ALT+Y`), the tool copies the file. If you click **No** (`ALT+N`), the tool cancels the popup without copying. The tool always moves files into the target configuration, except if a file is identical to the base or target version. In this case, the tool does not move the file.

Note: Do not edit a file version with `DO_NOT_EDIT` in its file name.

Configuration Merging Guidelines

The first milestone of an upgrade project is to generate the Java and SOAP APIs (by running `gwbc regen-java-api` and `gwbc regen-soap-api`) on the target release. To generate the Java and SOAP APIs, you must:

- Complete the merge of the data model. This includes all files in the `/extensions` and `/fieldvalidators` folders.
- Resolve issues encountered while trying to generate the APIs or start the QuickStart application server.

You can generate the Java and SOAP APIs even if you have errors in your enhancements, rules and PCF files.

Typical errors

- **Malformed XML** – The merge tool is not XML-aware. There might be occasions in which the file produced contains malformed XML. To check for well-formed XML, use free third-party tools such as Liquid XML, XML Marker, or Eclipse.
- **Duplicate typecodes** – As part of the merge process, you might have inadvertently merged in duplicate, matching typecodes.
- **Missing symmetric relationship on line-of-business-related typelists** – You might be missing a parent-child relationship with respect to the line-of-business-related typelist, as a result of merging.

After you have generated the Java and SOAP APIs, you can begin the work of upgrading integrations.

Second, after you can successfully generate the Java and SOAP APIs, work on starting the server.

In addition to the typical errors described previously, the server might fail to start due to cyclical graph reference errors. See “Identifying Data Model Issues” on page 53.

You can generate the APIs even if you have errors in your enhancements, rules and PCF files, although error messages will print upon server startup.

After the server can start on the target release, you can begin the database upgrade process.

Continue with the remainder of the configuration upgrade work, which includes evaluating existing PCF files and merging in desired changes.

Data Model Merging Guidelines

From a purely technical standpoint, not addressing the need to incorporate new features, the following are a few guidelines for merging the data model.

Merging Typelists – Overview

There is no automated process to merge typelists. This is a part of the merge process using the Configuration Upgrade Tool. In general, merge typelists before PCF files.

See the *Upgrade Diffs Report* for an inventory of differences in typekeys between the base release and the target release. To retrieve the *Upgrade Diffs Report* follow the procedure described in “Viewing Differences Between Base and Target Releases” on page 97.

Merge in Guidewire-provided typecodes related to lines of business and retire unused typecodes that you merge in. If you do not include these typecodes, you will have errors in any enhancements, rules, or PCF files that reference the typecode. This also simplifies the process for future upgrades as there will be fewer added line of business typecodes to review.

Pay particular attention if any Guidewire-provided typecodes have the same typecode as a custom version. In this case, modify one of the typecodes so they are unique. Contact Guidewire Support for details.

The Configuration Upgrade Tool displays most typelists you have edited in the CUSTOMER_EDIT filter. If your edits are simply additional typecodes, accept your version.

Use Guidewire Studio to verify PCF files, enhancements, and rules to identify any issues with the files and rules that reference typelists.

Merging Typelists – Simple Typelists

Merge in new typecodes from the target version, BillingCenter 8.0.4. If you do not merge the new typecode, you will have errors in any enhancements, rules, or PCF files that reference the typecode. If you do not want to use a new typecode, retire the typecode by setting the `retired` attribute to `true`.

Merging Typelists – Complex Typelists

A typecode can reference typecode values from another typelist using the `<category>` subelement. If a new typecode references an existing typecode, do not merge the new typecode unless the referenced typecode is retired. Otherwise, you are defining a new relationship. If the referenced typecode is also new, merge in both typecodes. If you do not want to use a new typecode, set the `retired` attribute for the typecode to `true`. The following table summarizes how to handle merging new typecodes that reference other typecodes:

Referenced typecode status	Action
new – exists only in target version	Merge in the new typecode and merge in the referenced typecode in its typelist. If you do not want to use the new typecode, retire it by setting the <code>retired</code> attribute of the typecode to <code>true</code> .
active – exists in base or custom version and is not retired	Do not merge the new typecode.
retired – exists in base or custom version and is retired	Merge in the new typecode. If you do not want to use the new typecode, retire it by setting the <code>retired</code> attribute of the typecode to <code>true</code> .

Reviewing Shared Typekey Configuration

As of version 8.0.3, BillingCenter enforces restrictions on the use of shared typekeys among subtypes.

Same Field Name and Typelist with Different Column

In BillingCenter 7.0 and earlier, if a shared typekey had the same field name and typelist, and specified a different column name, BillingCenter created only one of the typekey columns. The shared typekeys were stored in the single column. As of BillingCenter 8.0.3, if a shared typekey with the same field name and typelist specifies a different column name, BillingCenter creates different columns according to the specification. The database upgrade detects shared typekeys using a single column, creates the additional column, and moves the typekey data to the correct column.

Same Field and Column Names with Different Typelists

In BillingCenter 8.0.1 and 8.0.2, a typekey on subtypes could have the same field name and column name and reference different typelists. As of BillingCenter 8.0.3, this configuration is not allowed. The database upgrade reports an error if it detects this condition.

If you have subtypes with typekeys with the same field and column name that reference different typelists, update your data model configuration to use different column names for each typelist. The database upgrade then moves data to the new column to match the updated data model.

Merging Entity Extensions

BillingCenter 8.0.4 stores extensions in ETI and ETX files. An `Entity.eti` file defines a new entity. An `Entity.etx` file defines extensions to an existing entity.

Reviewing Optional Indexes

Guidewire often adds indexes to entities in the target configuration to improve the performance of database queries in BillingCenter 8.0.4. BillingCenter requires some of these indexes. Guidewire adds required indexes to entity definitions in the data model. Other indexes are recommended for most installations but can be disabled if they negatively impact performance. Guidewire adds optional indexes to entity extensions so you can disable any of these indexes if necessary.

Use the Configuration Upgrade Tool to resolve extension files. When you merge your custom extensions with Guidewire changes, review each new index added by Guidewire. In most cases, include the new index in the merged extension file. You can modify or remove index definitions based on usage in your deployment.

Reviewing Custom Extensions

Generate and review the data dictionary for the target version to identify any custom extensions that are now obsolete due to Guidewire adding a similar field to the base BillingCenter.

To generate the data dictionary

1. From the command line, navigate to the `bin` directory of the target version.
2. Run the command `gwbc regen-dictionary`.

This command generates the data and security dictionaries in the `build/dictionary` directory of the target version. To view the data dictionary, open `build/dictionary/data/index.html` in a web browser.

Compare the target version data dictionary with the version in your current environment. If you have extensions that are now available as base fields, consider migrating the data in those fields to the base version. Consider whether an extension is still on the appropriate entity. A new entity could be a more appropriate location for the extension. Review key data model changes that might impact your custom extensions.

If you change an extension location or migrate to a new base field, update any PCF, rule or library that references the extension to reference the new location.

Reconciling the Database with Custom Extensions

Extensions defined in ETI and ETX files must match the physical database. Delete all physical columns in the database that are not part of the base installation or defined as extensions before starting the server.

Merging Display Properties

The Configuration Upgrade Tool updates display properties files, such as `display.properties` to create a merged file with the extension `.merged`. You could have conflicts in the files if you have a different number of parameters for a key or if you have a different value.

If the number of parameters differs from the prior version, match your parameter set to the new version of the key.

If the value is different, choose which value you want to use in your BillingCenter configuration.

Merge changes into `display.properties.merged`. When you save the file, the Configuration Upgrade Tool saves it to the configuration module without the `.merged` extension.

If you have added locales, you can export a full list of display keys and typelists from the default BillingCenter 8.0.4 locale to any locale you have defined. This list includes a section for display keys and typelists that do not yet have values defined for your locale. You can use this list to determine which display keys and typelists require localized values. You can then specify those values and import the list. See “Translating New Display Properties and Typecodes” on page 47.

In BillingCenter 8.0, Studio trims trailing spaces from display keys by default. You can modify this behavior using the following procedure:

1. Click File → Settings.
2. Under IDE Settings click Editor.
3. Under Other, change the value of Strip trailing spaces on Save to None.
4. Click OK.

Upgrading Rules to BillingCenter 8.0.4

The Configuration Upgrade Tool does not upgrade rules. The tool classifies rules in the unmergeable filter. Within the target directory, Guidewire-provided default rules are located in `modules/configuration/config/rules`. The Configuration Upgrade Tool moves your custom rules to `modules/configuration/config/rules`.

Guidewire also copies the default rules for the current release to a BillingCenter 8.0.4 Rules folder within `modules/configuration/config/rules`. Use Studio to update your rules. You can use the rules in the BillingCenter 8.0.4 folder as a comparison. Compare your custom rules to the new default 8.0.4 versions and update your rules as needed.

You might find it useful to do a bulk comparison of default rules from the base release against the 8.0.4 versions to determine what types of changes Guidewire has made.

To compare rules between versions using the Rule Repository Report

1. If you want to compare default rules only, temporarily remove custom rules from your starting version by moving the `modules/configuration/config/rules` directory to a location outside the BillingCenter directory.
If you want to compare your custom rules against the BillingCenter 8.0.4 rules, do not move the `modules/configuration/config/rules` directory from your starting version. However, do remove the `BillingCenter<base version>` directory from `modules/configuration/config/rules/rules` of the starting version if this directory exists.
2. Open a command window.
3. Navigate to the `bin` directory of your starting version.
4. Enter the following command:
`gwbc regen-rulereport`
This command creates a rule repository report XML file in `build/rules`.
5. Append the starting version number to the XML file name.
6. Restore moved directories to the starting version.
7. Install files for a fresh BillingCenter 8.0.4 version. This is a separate configuration from the target configuration that you have merged. This version will only contain the default rules provided with BillingCenter 8.0.4.
8. Navigate to the `bin` directory of the new BillingCenter 8.0.4 version.
9. Enter the following command:
`gwbc regen-rulereport`

This command creates a rule repository report XML file in `build/rules`. There is a slight change to the path between the versions.

10. Append the target version number to the XML file name.
11. Open both rule report XML files in a merge tool. You do not merge base rules using the rule repository reports. However, looking at changes that Guidewire has made to the base rules can help you determine the types of changes you must make in your custom rules.
In your merge tool, disable whitespace differences and comments to reduce the amount of inconsequential differences shown between rules.

Update custom rules using Studio. Studio does not compare your rules directly with target rules. However, Studio provides powerful Gosu editing features not available in a standard text editor that can alert you to issues.

In Studio, you can compare custom rules to default BillingCenter 8.0.4 rules by opening the default rules in the BillingCenter 8.0.4 directory within `configuration → config → Rule Sets`. When you have finished updating all of your custom rules, delete the BillingCenter 8.0.4 rules directory from `modules/configuration/config/rules`.

Translating New Display Properties and Typecodes

BillingCenter 8.0.4 adds new display properties and typecodes. If you have defined additional locales, export these new display properties and typecodes to a file, define localized values, and reimport the localized values. If you do not have additional locales defined in your BillingCenter environment, skip this procedure.

To localize new display properties and typecodes

1. Export display keys by running the following command from your BillingCenter 8.0.4 environment `BillingCenter/bin` directory:
`gwbc export-l10ns -Dexport.file="translation_file" -Dexport.locale="language to export"`
2. Open the exported translation file in a text editor. The first section of the file lists display properties and typecodes that have a localized value. The second section lists display properties and typecodes that do not have a localized value.
3. Specify localized values for the untranslated properties.
4. Save the updated file.
5. Import the updated file by running the following command from your BillingCenter 8.0.4 environment `BillingCenter/bin` directory:
`gwbc import-l10ns -Dimport.file="translation_file" -Dimport.locale="language to import"`
After you import the localized typecodes and display keys, you can view them in Studio.

Validating the BillingCenter 8.0.4 Configuration

This topic includes procedures to validate the upgraded configuration.

Using Studio to Verify Files

You can use Studio to verify classes and enhancements, including libraries, PCF files, rules, and typelists without having to start BillingCenter. Do not start BillingCenter at this point. Studio can run without connecting to the application server.

To validate Studio resources

1. Start Guidewire Studio by running `gwbc studio` from the `BillingCenter\bin` directory.
2. Click **Analyze** → **Inspect Code....**
3. Set the **Inspection scope** to **module 'configuration'**.
4. Click **OK**. Studio runs inspections to identify incorrect Gosu syntax, issuing either a warning or an error.
5. Correct all identified errors with Studio. You can defer fixing warnings.

Starting BillingCenter and Resolving Errors

IMPORTANT In the process described in this section, do not point the BillingCenter server at a production database. The goal of this process is to test the configuration upgrade. Create an empty database account and point BillingCenter to this account for this process. See “Configuring the Database” on page 23 in the *Installation Guide* and “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.

Upon starting the server for the first time, you might receive errors that prevent the server from starting. In general, fixing errors and starting the server is an iterative process that involves:

1. Start the server for the target configuration.
BillingCenter encounters a configuration error and shuts down.
2. Copy the error message to a log file.
3. Locate the configuration causing the error, such as a line of code in a PCF.
4. Comment out the offending line.
After the server starts successfully, look at the log and start solving errors and introducing solutions into the environment. Assign errors to developers on your team.
5. Copy the commented file to the test bed for later analysis.
6. Begin again with step 1. Continue until the server starts successfully.

When the server starts successfully, resolve any remaining issues in the configuration that caused startup errors. Attempt to resolve each error individually and start the server to see if the fix worked.

Building and Deploying BillingCenter 8.0.4

After you apply and validate an upgrade to the configuration environment, rebuild and redeploy BillingCenter. Before you begin, make sure you have carefully prepared for this step. In particular, make sure you have updated your infrastructure appropriately.

Review this topic and then rebuild and redeploy BillingCenter to the application server. See “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide* of the target version for instructions.

WARNING Do not yet start BillingCenter. Only package the application file and deploy it to the application server. Starting BillingCenter begins the database upgrade.

If you have multiple Guidewire products, then upgrade, build, and deploy each individually before attempting to reintegrate them.

The Build Environment

With the exception of the database configuration, the first time you start the application server use the unmodified `config.xml` and `logging.properties` files provided with the target configuration. After the server starts successfully, you can merge in specific configurations of these files.

If you encounter build failures due to data dictionary generation, you can comment out this dictionary generation. Then, as you start the server, it reports any PCF configuration errors. After you have corrected PCF configurations, un-comment the dictionary generation and rebuild the application file.

Preserving JAR Files

Place custom JAR files in the `/config/lib` directory. Building and deploying a WAR or EAR file copies the JAR file into the appropriate place for it to be accessed by BillingCenter. JAR files in this location survive the upgrade process.

Upgrading the BillingCenter 8.0.x Database

This topic provides instructions for upgrading the BillingCenter database to BillingCenter 8.0.4.

If you are upgrading from a 7.0.x version, see “Upgrading the BillingCenter 7.0.x Database” on page 133 instead.

If you are upgrading from a 3.0.x version, see “Upgrading the BillingCenter 3.0.x Database” on page 249 instead.

This topic includes:

- “Upgrading Administration Data for Testing” on page 52
- “Identifying Data Model Issues” on page 53
- “Verifying Batch Process and Work Queue Completion” on page 54
- “Purging Data Prior to Upgrade” on page 54
- “Validating the Database Schema” on page 55
- “Checking Database Consistency” on page 56
- “Creating a Data Distribution Report” on page 56
- “Generating Database Statistics” on page 57
- “Creating a Database Backup” on page 58
- “Updating Database Infrastructure” on page 58
- “Preparing the Database for Upgrade” on page 58
- “Setting Linguistic Search Collation” on page 59
- “Field Encryption and the Upgraded Database” on page 60
- “Customizing the Upgrade” on page 60
- “Running the Commission Payable Calculations Process” on page 71
- “Configuring the Database Upgrade” on page 71

- “Checking the Database Before Upgrade” on page 78
- “Disabling the Scheduler” on page 78
- “Suspending Message Destinations” on page 79
- “Starting the Server to Begin Automatic Database Upgrade” on page 79
- “Viewing Detailed Database Upgrade Information” on page 83
- “Dropping Unused Columns on Oracle” on page 83
- “Exporting Administration Data for Testing” on page 84
- “Final Steps After The Database Upgrade is Complete” on page 85

Upgrading Administration Data for Testing

You might want to create an upgraded administration data set for development and testing of rules and libraries with BillingCenter 8.0.4. You can wait until the full database upgrade is complete and then export the administration data, as described in “Exporting Administration Data for Testing” on page 84. Or, you can upgrade only the administration data to have this data available earlier in the upgrade process. Use the procedure in this section to create an upgraded administration data set before upgrading the full database.

To upgrade administration data

1. Export administration data from your current (pre-upgrade) BillingCenter production instance:
 - a. Log on to BillingCenter as a user with the `viewadmin` and `soapadmin` permissions.
 - b. Click the **Administration** tab.
 - c. Choose **Import/Export Data**.
 - d. Select the **Export** tab.
 - e. Select **Admin** from the **Data to Export** dropdown.
 - f. Click **Export**. BillingCenter exports an `admin.xml` file.
2. On a new pre-upgrade development environment based on your production configuration, create an empty version of `importfiles.txt` in the `modules/configuration/config/import/gen` directory.
3. Create empty versions of the following CSV files:
 - `activity-patterns.csv`
 - `authority-limits.csv`
 - `reportgroups.csv`
 - `roleprivileges.csv`
 - `rolereportprivileges.csv`Leave `roles.csv` as the original complete file.
4. Start the development environment server by opening a command prompt to `BillingCenter/bin` and entering the following command:
`gwbc dev-start`
5. Import this administration data into the development environment.
 - a. Log on to BillingCenter as a user with the `viewadmin` and `soapadmin` permissions.
 - b. Click the **Administration** tab.
 - c. Choose **Import/Export Data**.
 - d. Select the **Import** tab.

- e. Click **Browse....**
- f. Select the `admin.xml` file that you exported in step 1.
- g. Click **Open**.
6. Create a backup of the new development environment database.
7. Create a new database account for the development environment on a database management system supported by BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <http://guidewire.custhelp.com>.
See “Configuring the Database” on page 23 in the *Installation Guide* for instructions to configure the database account.
8. Restore the backup of the database containing the imported administration data into the new database.
9. Connect your upgraded target BillingCenter 8.0.4 configuration to the restored database.
10. Start the BillingCenter 8.0.4 server to upgrade the database.
11. Export the upgraded administration data:
 - a. Start the BillingCenter 8.0.4 server by navigating to `BillingCenter/bin` and running the following command:
`gwbc dev-start`
 - b. Open a browser to BillingCenter 8.0.4.
 - c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
 - d. Click the **Administration** tab.
 - e. Choose **Import/Export Data**.
 - f. Select the **Export** tab.
 - g. For **Data to Export**, select **Admin**.
 - h. Click **Export**. Your browser will note that you are opening a file and will prompt you to save or download the file.
 - i. Select to download the `admin.xml` file. You can import this XML file into local development environments of BillingCenter 8.0.4.

Identifying Data Model Issues

Before you upgrade a production database, identify issues with the data model by running the database upgrade on an empty database. This process does not identify all possible issues. The database upgrade does not detect issues caused by specific data in your production database. Instead, this procedure identifies issues with the data model.

Complete the following procedure to identify data model issues, and correct any issues on an empty schema. Then, follow the full list of procedures in this topic to upgrade a production database. This list begins with “Verifying Batch Process and Work Queue Completion” on page 54 and finishes with “Final Steps After The Database Upgrade is Complete” on page 85.

To identify data model issues

1. Create an empty schema of your starting version database. You can do this in a development environment by pointing the development BillingCenter installation at an empty schema and starting the BillingCenter server. See “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.
2. Complete the configuration upgrade for data model files in your starting version, according to the instructions in “Upgrading the BillingCenter 8.0.x Configuration” on page 29. You do not need to complete the merge process for all files.
3. Configure your upgraded development environment to point to the database account containing the empty schema of your old version. See “Configuring a Database Connection” on page 62 in the *Installation Guide*.
4. Start the BillingCenter server in your upgraded development environment. The server performs the database upgrade to BillingCenter 8.0.4. See “Starting the Server to Begin Automatic Database Upgrade” on page 79.
5. Check for errors reported during the upgrade process. Resolve any issues before upgrading your production database. You can use the `IDatabaseUpgrade` plugin to run custom SQL before and after the database upgrade. For more information, see “Running Custom Version Checks and Triggers” on page 61.

Verifying Batch Process and Work Queue Completion

All batch processes and work queues must complete before beginning the upgrade. Check the status of batch processes and work queues in your current production environment.

To check the status of batch processes and work queues

1. Log in to BillingCenter as the superuser.
2. Press Alt + Shift + T. BillingCenter displays the **Server Tools** tab.
3. Click **Batch Process Info**.
4. Select **Any** from the **Processes** drop-down filter.
5. Click **Refresh**.
6. Check the **Status** column for each batch process listed. This list also includes batch processes that are writers for distributed work queues. If any of the batch processes have a **Status** of **Active**, wait for the batch process to complete before continuing with the upgrade.

Purging Data Prior to Upgrade

This topic includes recommendations for purging certain types of data from the database prior to upgrade. Removing unused records can improve the performance of the database upgrade and BillingCenter.

Purging Old Messages from the Database

Purge completed inactive messages before upgrading the database. Doing so reduces the complexity of the database upgrade.

You can use the following command from the current (pre-upgrade) customer configuration `admin/bin` directory to purge completed messages from the `bc_MessageHistory` table:

```
messaging_tools -password password -server http://server:port/instance -purge MM/DD/YYYY
```

This tool deletes completed messages with a send time before the date `MM/DD/YYYY`.

Or, you can use the following web service API:

```
IMessageToolsAPI.purgeCompletedMessages(java.util.Calendar cutoff)
```

Periodically purge old messages to prevent the database from growing unnecessarily.

Purge messages from the database before starting BillingCenter, so the database upgrade does not attempt to convert those rows.

You cannot resend old messages after the upgrade. This is because integrations change and the message payload might be different. It is important that messages that have failed or not yet been consumed finish prior to upgrading.

After you purge completed inactive messages, reorganize the `bc_MessageHistory` table. You might also want to rebuild any indexes on the table. Contact Guidewire Support if you need assistance.

Purging Completed Workflows and Workflow Logs

Each time BillingCenter creates an activity, the activity is added to the `bc_Workflow`, `bc_WorkflowLog` and `bc_WorkflowWorkItem` tables. Once a user completes the activity, BillingCenter sets the workflow status to completed. The `bc_Workflow`, `bc_WorkflowLog` and `bc_WorkflowWorkItem` table entry for the activity are never used again. These tables grow in size over time and can adversely affect performance as well as waste disk space. Excessive records in these tables also negatively impacts the performance of the database upgrade.

Remove workflows, workflow log entries, and workflow items for completed activities to improve database upgrade and operational performance and to recover disk space.

BillingCenter includes work queues to purge completed workflows and their logs that are older than a configurable number of days. Guidewire recommends that you purge completed workflows and their logs periodically. This reduces performance issues caused by having a large number of unused workflow log records.

To set the number of days after which the `purgeworkflows` process purges completed workflows and their logs, set the following parameter in `config.xml`:

```
<param name="WorkflowPurgeDaysOld" value="value" />
```

Set the value to an integer. By default, `WorkflowPurgeDaysOld` is set to 60. This is the number of days since the last update to the workflow, which is the completed date.

You can launch the Purge Workflows batch process from the `BillingCenter/admin/bin` directory with the following command:

```
maintenance_tools -password password -startprocess PurgeWorkflows
```

You can also purge only the logs associated with completed workflows older than a certain number of days. Run the `purgeworkflowlogs` process instead. This process leaves the workflow records and removes only the workflow log records. The `purgeworkflowlogs` process is configured using the `WorkflowLogPurgeDaysOld` parameter rather than `WorkflowPurgeDaysOld`.

You can launch the Purge Workflow Logs batch process from the `BillingCenter/admin/bin` directory with the following command:

```
maintenance_tools -password password -startprocess PurgeWorkflowLogs
```

Validating the Database Schema

This validation detects the unlikely event that the data model defined by your configuration files has become out of sync with the database schema. While the pre-upgrade server is running, use the `system_tools` command in `admin/bin` of the customer configuration to verify the database schema:

```
system_tools -password password -verifydbschema -server servername:port/instance
```

Correct any validation problems in the database before proceeding. Contact Guidewire Support for assistance.

Following the database upgrade, run this command again from the `admin/bin` directory of the target (upgraded) configuration.

Checking Database Consistency

BillingCenter has hundreds of internal database consistency checks. Before upgrading, run consistency checks to verify the integrity of your data.

Run database consistency checks early in the upgrade project. Fix any consistency errors. Continue to periodically run consistency checks and resolve issues so that your database is ready to upgrade when you begin the upgrade procedure. Consistency issues might take some time to resolve, so begin the process of running consistency checks and fixing issues early. Contact Guidewire Support for information on how to resolve any consistency issues.

After the database upgrade, run consistency checks again from the BillingCenter **Consistency Checks** page.

To run consistency checks

1. Start the BillingCenter server if it is not already running.
2. Log in to BillingCenter with an administrator account.
3. Press Alt + Shift + T to access the **Server Tools**.
4. Click **Info Pages**.
5. Select **Consistency Checks** from the drop-down list.
6. To increase the number of threads used to run consistency checks, increase the **Number of threads**. The number of threads to use depends on the capability of your database server. Increasing the number of threads can improve performance of consistency checks as long as your server can process the threads. Guidewire recommends starting with five threads. If too many threads are used, there is a greater chance that current users experience reduced performance if the database server is fully loaded.

To set the number of threads in versions prior to 8.0, specify a value for the `checker.threads` parameter within the database block of `config.xml`.

```
<database>
  ...
  <param name="checker.threads" value="5" />
  ...
</database>
```

7. Click **Run Consistency Checks**.

See also

“[Checking Database Consistency](#)” on page 36 in the *System Administration Guide*.

Creating a Data Distribution Report

Generate a data distribution report for the database before an upgrade. Save the output of this report. Run the report again after the upgrade to ensure the distribution is still correct.

Guidewire is very interested in the data distribution of your databases. Guidewire uses these reports to better understand the nature of your database and to optimize BillingCenter performance. Guidewire appreciate copies of your reports, both before and after upgrades.

You can also use this information to tune the application server cache. See “Application Server Caching” on page 63 in the *System Administration Guide*.

To create a database distribution report

1. In config.xml, set <param name="EnableInternalDebugTools" value="true"/>.
2. Start the BillingCenter application server.
3. Log into BillingCenter as an administrative user.
4. Type ALT + SHIFT + T while in any screen to reach the **Server Tools** page.
5. Choose **Info Pages** from the **Server Tools** tab.
6. Choose the **Data Distribution** page from the **Info Pages** dropdown.
7. Enter a reason for running the Data Distribution batch job in the **Description** field.
8. On this page, select the **Collect distributions for all tables** radio button and check all checkboxes to collect all distributions.
9. Push the **Submit Data Distribution Batch Job** button on this page to start the data collection.
10. Return to the **Data Distribution** page and push its **Refresh** button to see a list of all available reports. The batch job has completed when the **Available Data Distribution** list on the **Data Distribution** page includes your description.
11. Select the desired report and use the **Download** button to save it zipped to a text file. Unzip the file to view it.

Generating Database Statistics

To optimize the performance of the BillingCenter database, it is a good idea to update database statistics on a regular basis. Both SQL Server and Oracle can use these statistics to optimize database queries.

If you update database statistics on a regular basis, you do not need to update statistics before an upgrade. If you do not update database statistics on a regular basis, Guidewire recommends that you update incremental statistics before running the upgrade.

To generate incremental database statistics

1. Get the proper SQL statements for updating the statistics in BillingCenter tables by running the following command, depending on your starting version:

For upgrades from 8.0.0:

```
maintenance_tools -getincrementaldbstatisticsstatements -password password  
-server http://server:port/instance > db_stats.sql
```

For upgrades from 8.0.1 or newer:

```
system_tools -password password -getincrementaldbstatisticsstatements -server  
http://server:port/instance > db_stats.sql
```

2. Run the resulting SQL statements against the BillingCenter database.

You can configure SQL Server to periodically update statistics using SQL. See your database documentation and “Configuring Database Statistics” on page 39 in the *System Administration Guide* for more information.

The database upgrade can take a long time, and has built-in statistics collection that help you see if any part of the upgrade is slow. Collect these statistics, and compare them to the statistics you collected before the upgrade. The config.xml file has parameters that control this statistics collection.

If you disabled statistics collection during upgrade by setting updatestatistics to false, Guidewire recommends that you generate full statistics as soon as possible after the upgrade. For instructions, see “Commands for Updating Database Statistics” on page 40 in the *System Administration Guide*.

Creating a Database Backup

Prepare the environment so that you can make a total recovery of the original installation if you run into problems during the upgrade.

The first time you start the BillingCenter server after running the upgrade tool, the server updates the database. During its work, the database upgrader minimizes the logging that it does. For these reasons, back up your database before starting an upgrade. Your pre-upgrade database might not be recoverable after an upgrade.

Updating Database Infrastructure

Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

Preparing the Database for Upgrade

This topic notes steps to prepare the database for the upgrade process.

Ensuring Adequate Free Space

The database upgrade requires significant free space. Make sure the database has at least 50% of the current database size available as free space.

Disabling Replication

Disable database replication during the database upgrade.

Assigning Default Tablespace (Oracle only)

Set the default tablespace for the database user to the one mapped to the logical tablespace OP in config.xml.

The database upgrade creates temporary tables during the upgrade without specifying the tablespace. If the Oracle database user was created without a default tablespace, Oracle by default creates the tables in the SYSTEM tablespace. The Guidewire database user is likely not to have the required quota permission on the SYSTEM tablespace. This results in an error of the type:

```
java.sql.SQLException: ORA-01950: no privileges on tablespace 'SYSTEM'
```

Even if the default tablespace is not SYSTEM, if the Guidewire database user does not have quota permission on the default tablespace, the temporary table creation during upgrade fails.

Setting Linguistic Search Collation

WARNING For SQL Server, compare the default collation of the database to the collation defined for your locale. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your SQL Server database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This attribute change results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, this adds time to the total database upgrade process.

WARNING Oracle Java Virtual Machine (JVM) must be installed on all Oracle databases hosting BillingCenter. The only exception is when the BillingCenter application locale is English and you only require case-insensitive searches. Ensure that Oracle initialization parameter `java_pool_size` is set to a value greater than 50 MB.

You can specify how you want BillingCenter to collate search results. The `strength` attribute of the `LinguisticSearchCollation` element of `GWLanguage` in `language.xml` in the currently selected region specifies how BillingCenter sorts search results. You can set the `strength` to `primary` or `secondary`.

With `LinguisticSearchCollation strength` set to `primary`, BillingCenter searches results in a case-insensitive and accent-insensitive manner. BillingCenter considers an accented character equal to the unaccented version of the character if the `LinguisticSearchStrength` for the default application locale is set to `primary`. For example, with `LinguisticSearchCollation strength` set to `primary`, BillingCenter treats “Renée”, “Renée”, “renée” and “reneé” the same.

With `LinguisticSearchCollation strength` set to `secondary`, BillingCenter searches results in a case-insensitive, accent-sensitive manner. BillingCenter does not consider an accented character equal to the unaccented version of the character if the `LinguisticSearchCollation strength` for the default application locale is set to `secondary`. For example, with `LinguisticSearchCollation strength` set to `secondary`, a BillingCenter search treats “Renée” and “renée” the same but treats “Renée” and “reneé” differently. By default, BillingCenter uses a `LinguisticSearchCollation strength` of `secondary`, which specifies case-insensitive, accent-sensitive searching.

The `collations.xml` file defines the collations to use for different locales and different collation strengths. The `primary`, `secondary`, and `tertiary` attributes of the `Collation` element define the collation to use depending on the `LinguisticSearchCollation strength` attribute in `language.xml`.

BillingCenter 7.0 introduced configurable linguistic searching for SQL Server databases. In releases prior to BillingCenter 7.0, BillingCenter used the collation setting of the database server. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This attribute change results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, dropping and recreating indexes adds time to the total database upgrade process.

For sorting search results, the following rules apply:

- **Case** – All searches ignore the case of the letters, whether `LinguisticSearchCollation strength` is set to `primary` or `secondary`. “McGrath” equals “mcgrath”.
- **Punctuation** – Punctuation is always respected, and never ignored. “O'Reilly” does not equal “OReilly”.
- **Spaces** – Spaces are respected. “Hui Ping” does not equal “HuiPing”.
- **Accents** – An accented character is considered equal to the unaccented version of the character if `LinguisticSearchCollation strength` is set to `primary`. An accented character is not equal to the unaccented version if `LinguisticSearchCollation strength` is set to `secondary`.

Japanese only

- **Half Width/Full Width** – Searches under a Japanese region always ignore this difference.
- **Small/Large Kana** – Japanese small/large letter differences are ignored only when `LinguisticSearchCollation strength` is set to `primary`, meaning accent-insensitive.
- **Katakana/Hiragana sensitivity** – Searches under a Japanese region always ignore this difference.
- The long dash character is always ignored.
- Soundmarks (` and °) are only ignored if `LinguisticSearchCollation strength` is set to `primary`.

German only

- Vowels with an umlaut compare equally to the same vowel followed by the letter e. Explicitly, “ä”, “ö”, “ü” are treated as equal to “ae”, “oe” and “ue”.
- The Eszett, or sharp-s, character “ß” is treated as equal to “ss”.

BillingCenter populates denormalized values of searchable columns to support the search collation. For example, with `LinguisticSearchCollation strength` set to `primary`, BillingCenter stores the value “Renée”, “Renée”, “reneé” and “reneé” in a denormalized column as “reneé”. With `LinguisticSearchCollation strength` set to `secondary`, BillingCenter stores a denormalized value of “reneé” for “Renée” or “reneé” and stores “renée” for “Renée” or “reneé”. Japanese and German locales make additional changes when storing values in denormalized columns in order to conform to the rules listed previously for those locales.

Any time you change the `LinguisticSearchCollation strength` and restart the server, BillingCenter repopulates the denormalized columns. Previous versions of BillingCenter populated the denormalized columns with lowercase values for case-insensitive search, equivalent to setting `LinguisticSearchCollation strength` to `secondary`. If you set `LinguisticSearchCollation strength` to `primary`, BillingCenter repopulates the denormalized columns, substituting any accented characters for their base equivalents. This process can take a long time, depending on the amount of data. Therefore, if you want to change `LinguisticSearchCollation strength` to `primary`, you might want to do so after the database upgrade. If you are concerned about the duration of the database upgrade, you can change your search collation settings after the upgrade. During a maintenance period, change `LinguisticSearchCollation strength` to `primary` and restart the server to repopulate the denormalized columns.

For Japanese locales, the BillingCenter database upgrade from a prior major version repopulates the denormalized columns regardless of the `LinguisticSearchCollation strength` value. BillingCenter must repopulate the denormalized columns for Japanese locales to have search results obey the Japanese-only rules listed previously.

For more information, see “Linguistic Search and Sort” on page 159 in the *Globalization Guide*.

Field Encryption and the Upgraded Database

The database upgrader handles most encrypted fields with no problem. If you have started to use field encryption and have changed fields in the database from no field encryption to encrypted, specifying some encryption algorithm, the upgrader preserves this encryption. However, if you later change the encryption to another algorithm type, the upgrader does not handle this case. See “Changing Your Encryption Algorithm Later” on page 258 in the *Integration Guide*.

Customizing the Upgrade

The `IDataModelUpgrade` plugin interface provides hooks for custom code that you want to run during the database upgrade. You can use the `IDataModelUpgrade` plugin to:

- execute custom version checks to test data or the data model itself before starting the upgrade.

- make custom database changes before or after the database upgrade.
- make data model changes to archived entities.

For example, you might fix a consistency check failure issue, correct issues reported by version checks, or delete a custom extension that you are no longer using.

IMPORTANT BillingCenter 3.0 included a similar plugin interface, `IDatabaseUpgrade`. If you previously implemented `IDatabaseUpgrade` for an upgrade to BillingCenter 3.0, you must now implement `IDatamodelUpgrade` if you want to execute custom upgrade code.

Running Custom Version Checks and Triggers

You can use the `IDatamodelUpgrade` plugin to run custom version checks and triggers before and after the database upgrade. The `IDatamodelUpgrade` plugin interface contains method signatures for two methods that you must define in your plugin. These signatures are:

- `property getBeforeUpgradeDatamodelChanges() : List<IDataModelChange<BeforeUpgradeVersionTrigger>>`
- `property getAfterUpgradeDatamodelChanges() : List<IDataModelChange<AfterUpgradeVersionTrigger>>`

Each method returns a list of `IDataModelChange` entities, each taking a `BeforeUpgradeVersionTrigger` or `AfterUpgradeVersionTrigger` type parameter. The `IDataModelChange` interface has two methods that you use to make data model changes. The `getDatabaseUpgradeVersionTrigger` method is for changes to the database. The `getArchivedDocumentUpgradeVersionTrigger` method is for changes to archived entities. If your organization has not implemented archiving or you do not want to make changes to archived entities, return null for `getArchivedDocumentUpgradeVersionTrigger`.

The `getAfterUpgradeDatamodelChanges` method runs after the Guidewire upgrade version triggers. You can use this method to move data into extension tables or columns that did not exist prior to upgrading.

You can return an empty list from either `getBeforeUpgradeDatamodelChanges` or `getAfterUpgradeDatamodelChanges`. For example, if you only have triggers to run before the upgrade, you can return an empty list from `getAfterUpgradeDatamodelChanges`.

Modifying Tables

Both `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` base classes provide a protected `getTable` method that accepts a `string` parameter. The `getTable` method returns an `IBeforeUpgradeTable` or `IAfterUpgradeTable` object that provides a number of methods for DDL and DML operations, such as:

- `create` – Create the table if it does not already exist. The table must be related to an entity defined in the data model. This method is available only for `IBeforeUpgradeTable`.
- `delete` - Deletes rows from a table. Returns a builder (`IBeforeUpgradeDeleteBuilder` for `IBeforeUpgradeTable`, `IDeleteBuilder` for `IAfterUpgradeTable`) that has methods for comparing data to restrict which rows are deleted.
- `drop` - Drops the table.
- `dropColumns` - Drops multiple columns from the table.
- `getColumn` – Returns an `IBeforeUpgradeColumn` or `IAfterUpgradeColumn` object that has methods to perform DDL operations on the column such as create, drop, rename, and more.
- `insert` – Returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable`) to perform an insert operation.
- `insertSelect` – Returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`) for SQL to perform an insert operation using data selected from a table.

- `rename` - Renames the table.
- `update` – Returns a builder (`IBeforeUpgradeUpdateBuilder` for `IBeforeUpgradeTable`, `IUpdateBuilder` for `IAfterUpgradeTable`) for SQL to perform an update operation.

For DML operations, call the `execute` method on the builder to actually perform the operation. The `execute` method runs in its own transaction. You do not need to handle transactions and `TransactionManager`.

There are more methods on the `IBeforeUpgradeTable` and `IAfterUpgradeTable` classes documented in the Guidewire Gosu API documentation. To generate the Guidewire Gosu API documentation, run the `gwbc regen-gosudoc` command from the BillingCenter `bin` directory. Then, open `BillingCenter/build/gosudoc/index.html`.

The methods for `BeforeUpgradeVersionTrigger` intentionally take strings but not entities or properties. This is because the name of the column could change in the future. Consider `PropertyA` on `EntityE` which corresponds to column A in the database. Suppose you use `PROPERTYA_PROP` in a version trigger at minor version 200, but at minor version 250, you decide to rename the backing column from A to B. The version trigger you wrote in the past would break because it would execute before the rename operation and would try to use the new column name.

`AfterUpgradeVersionTrigger` is very similar to `BeforeUpgradeVersionTrigger`. A few differences include:

- The `AfterUpgradeVersionTrigger` DML builders use the query builder, `IQueryBuilder`.
- In an `AfterUpgradeVersionTrigger` you can use properties and types in addition to strings.
- Some DDL operations are not provided on the `IAfterUpgradeTable` object, including creating a table or adding a column.

Unless you require one of the unique capabilities of `AfterUpgradeVersionTrigger`, use `BeforeUpgradeVersionTrigger` for custom version triggers.

Upgrading Typelists

The `BeforeUpgradeVersionTrigger` class includes a `getTypeKeyID` method with the following signature:

```
protected final Integer getTypeKeyID(IEntityType subtype)
```

Note: Protected methods do not appear in the Gosu documentation. Use CTRL + SPACE in Studio to show available methods and properties.

The `getTypeKeyID` method returns the integer ID of the type code in the type list matching the given table name. This method checks both the existing typelist tables and the metadata files to determine what all typekey IDs will be after upgrade. Therefore, the `getTypeKeyID` method works as expected even before a new typekey or typelist table is created during the automatic schema upgrade phase.

This method also works for orphaned typecodes that have not yet been removed from the database. These are typecodes that still exist in the database table but not in the metadata file. You can use the `getTypeKeyID` method for remapping usages of orphaned typecodes.

Version Checks

In some cases, you might want to check for a certain condition in the database before the upgrade proceeds. This is referred to as a version check. Only read operations are available in version checks. For example, you can write a version check to query a table or check the existence of a table or column, but the check cannot insert new rows. The `BeforeUpgradeVersionTrigger` class includes a `hasVersionCheck` method that you must define to return true or false. If the trigger does include a version check, overwrite the `createVersionCheck` method to define your custom version check. For standalone version checks that are not associated with a version trigger, you can use `BeforeUpgradeVersionCheckWrapper`.

The upgrade executes all custom version checks before custom version triggers. The upgrade runs Guidewire version checks after all custom `BeforeUpgradeVersionTrigger` implementations, so you can create a `BeforeUpgradeVersionTrigger` to correct issues detected by the Guidewire version checks.

If a custom version check fails, the upgrade stops before running any upgrade triggers. Correct the issue and restart the upgrade.

Order of Execution

The upgrade performs actions in the following order:

Step	Action	In the event of failure due to a data issue...
1	Custom version checks	Correct the data issue. Restart the upgrade. You do not need to restore the database because the upgrade has not made any changes.
2	Custom <code>BeforeUpgradeVersionTrigger</code> implementations	Restore the database from a backup. Correct the data issue. Consider adding custom version checks to test for other instances of the data issue.
3	Guidewire version checks	If you do not have any custom <code>BeforeUpgradeVersionTrigger</code> implementations, correct the data issue and restart the upgrade. If you do have custom <code>BeforeUpgradeVersionTrigger</code> implementations, restore the database from a backup. Then, correct the data issue. In either case, consider creating a custom <code>BeforeUpgradeVersionTrigger</code> implementation to correct the data issue.
4	Guidewire version triggers	A failure due to data issues at this stage is unlikely. Contact Guidewire Support.
5	Automated data model upgrade to update the database to the defined data model.	A failure due to data issues at this stage is unlikely. Contact Guidewire Support.
6	Guidewire version triggers that require the updated data model in the database	A failure due to data issues at this stage is unlikely. Contact Guidewire Support.
7	Custom <code>AfterUpgradeVersionTrigger</code> implementations	Restore the database from a backup. Correct the data issue. Consider creating a custom <code>BeforeUpgradeVersionTrigger</code> implementation to correct the data issue if possible.

The preceding table describes failure cases that are caused by data issues. If the upgrade fails for other reasons, such as a disruption of the database server, fix the issue causing the disruption, restore the database, and restart the upgrade.

Versioning

Each `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` instance requires a minor version number, passed as an integer. If the data model version number is less than or equal to the number passed to the instance, then the trigger executes. Whenever you make a data model change, or you want to force an upgrade, increment the version number in `extensions.properties`.

To run custom version checks and triggers

1. Create a new package, such as `companyName.upgrade`, to store your custom version triggers.
 - a. Open Studio.
 - b. In the Studio Project window, expand `configuration`.
 - c. Right-click `gsrc` and click `New → Package`.
 - d. Enter a package name for upgrade purposes, such as `companyName.upgrade`.
2. Right-click the upgrade package and click `New → Gosu Class`.
3. Enter a name for the class and click `OK`.

- 4.** Create a new Gosu class that extends `CustomerDatamodelUpgrade` and implements `IDatamodelUpgrade`. The class you create must define the `getBeforeUpgradeDatamodelChanges` and `getAfterUpgradeDatamodelChanges` methods. This class is the container from which you call custom version trigger classes.

For example:

```
package companyName.upgrade
uses gw.plugin.upgrade.IDatamodelUpgrade
uses java.lang.Iterable
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger
uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger
uses java.util.ArrayList
uses gw.api.datamodel.upgrade.CustomerDatamodelUpgrade
uses gw.api.datamodel.upgrade.IDatamodelChange
uses gw.api.database.upgrade.DatamodelChangeWithoutArchivedDocumentChange

class TestDatamodelUpgradeImpl extends CustomerDatamodelUpgrade implements IDatamodelUpgrade {

    override property get BeforeUpgradeDatamodelChanges() :
        List<IDatamodelChange<BeforeUpgradeVersionTrigger>> {
        var list = new ArrayList<IDatamodelChange<BeforeUpgradeVersionTrigger>>()
        list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger1()))
        list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger2()))
        return list
    }

    override property get AfterUpgradeDatamodelChanges() :
        List<IDatamodelChange<AfterUpgradeVersionTrigger>> {
        var list = new ArrayList<IDatamodelChange<AfterUpgradeVersionTrigger>>()
        list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new AfterVersionTrigger1()))
        return list
    }
}
```

- 5.** Create your custom `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` Gosu classes. See “`IDatamodelUpgrade` API Examples” on page 64.

- 6.** Implement the `IDatamodelUpgrade` plugin with the new class.

- a. Start Guidewire Studio 8.0.4 by entering `gwbc studio` from the `BillingCenter/bin` directory.
- b. In Studio, expand `configuration` → `config` → `Plugins`.
- c. Right-click `registry` and click `New` → `Plugin`.
- d. In the `Plugin` dialog, enter the name `IDatamodelUpgrade`. For this plugin, the name must match the interface.
- e. In the `Plugin` dialog, click the ... button.
- f. In the `Select Plugin Class` dialog, type `IDatamodelUpgrade` and select the `IDatamodelUpgrade` interface.
- g. In the `Plugin` dialog, click `OK`. Studio creates a `GWP` file under `Plugins` → `registry` with the name you entered.
- h. Click the `Add Plugin` icon (a plus sign) and select `Add Gosu Plugin`.
- i. For `Gosu Class`, enter your class, including the package.
- j. Save your changes.

When you start the server to perform the database upgrade from a prior major version, the upgrade calls the plugin and runs your custom methods.

`IDatamodelUpgrade` API Examples

This topic first introduces the basic structure of a `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger`. Next it shows methods that can be included within the `execute` method of these triggers to modify the database.

This topic includes:

- “BeforeUpgradeVersionTrigger Structure” on page 65
- “AfterUpgradeVersionTrigger Structure” on page 66
- “Altering Columns to Match Data Model” on page 66
- “Altering a Non-nullable Column to Nullable” on page 66
- “Creating Columns” on page 67
- “Dropping Columns” on page 68
- “Renaming Columns” on page 68
- “Setting a Column Value for a Specific Subtype” on page 68
- “Creating Tables” on page 69
- “Renaming Tables” on page 69
- “Deleting Rows” on page 69
- “Inserting Rows” on page 70
- “Inserting Data Selected from Another Table” on page 70
- “Updating Rows” on page 71

BeforeUpgradeVersionTrigger Structure

A custom BeforeUpgradeVersionTrigger subclass has the following structure.

```
package companyName.upgrade.before

uses gw.api.database.upgrade.before.BeforeUpgradeVersionCheck
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger

class myBeforeUpgradeTrigger extends BeforeUpgradeVersionTrigger {

    construct() {
        super(dataModelVersionNumber)
    }

    override function execute() {
        // Perform actions here.
    }

    override function hasVersionCheck() : boolean {
        // return true if creating a version check to determine whether the trigger can run.
        // return false if you are not implementing a version check.
    }

    override property get Description() : String {
        return "Description of the version trigger."
    }

    // Override the createVersionCheck method if you are implementing a version check.
    override function createVersionCheck() : BeforeUpgradeVersionCheck {
        return new BeforeUpgradeVersionCheck(dataModelVersionNumber) {

            override function verifyUpgradability() {
                if (condition to detect) {
                    addVersionCheckProblem("description of issue")
                }
            }

            override property get Description() : String {
                return "Description of the version check."
            }
        }
    }
}
```

Define the execute method to perform the actions you want your custom trigger to perform. Some examples are provided in subsequent topics.

AfterUpgradeVersionTrigger Structure

A custom AfterUpgradeVersionTrigger subclass has the following structure.

```
package companyName.upgrade.after

uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger

class myAfterUpgradeTrigger extends AfterUpgradeVersionTrigger{

    construct() {
        super(dataModelVersionNumber)
    }

    override function execute() {
        // Perform actions here.
    }

    override property get Description(): String {
        return "Description of the version trigger."
    }
}
```

Altering Columns to Match Data Model

In most cases, you do not need to alter a column to match a change to the column type in the logical data model. The upgrader automatically applies data model changes to the database. However, this occurs after all custom BeforeUpgradeVersionTrigger instances have run, so Guidewire provides methods to alter database columns to match the data model.

Altering a single column

If you need to alter a single column for use in a BeforeUpgradeVersionTrigger, modify the data model file, then use the alterColumnTypeToMatchDatamodel method of IBeforeUpgradeColumn.

For example:

```
var table = getTable("TableName")
var column = table.getColumn("ColumnName")
column.alterColumnTypeToMatchDatamodel()
```

Altering multiple column

To alter multiple columns for use in a BeforeUpgradeVersionTrigger, modify the data model file, then use the alterMultipleColumnsToMatchDatamodel method of IBeforeUpgradeTable.

For example:

```
var table = getTable("TableName")
var columnsToChange = new IBeforeUpgradeColumn[2]

columnsToChange[0] = table.getColumn("column1")
columnsToChange[1] = table.getColumn("column1")

table.alterMultipleColumnsToMatchDatamodel(columnsToChange)
```

Altering a Non-nullable Column to Nullable

To alter a column from non-nullable to nullable, use the IBeforeUpgradeColumn method alterColumnToNullable.

For example:

```
var table = getTable("TableName");
table.getColumn("ColumnName").alterColumnToNullable();
```

Creating Columns

The database upgrader automatically creates a column that is added to the data model if the column meets one of the following criteria:

- Nullable
- Non-nullable with a default value specified in the metadata
- Non-nullable without a default value if there are no rows in the table
- The column is an editable field

However, you might want to explicitly create the column in your upgrade trigger if you want the trigger to perform an action on the column such as populating it.

In the data model, the column must be defined as a property on an entity. The database upgrade will determine the correct datatype and nullability from the data model.

Creating a new column is moderately expensive in terms of performance of the upgrade.

Creating a Column

To create a column, invoke the `create` method on the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")  
  
// Create column with given name.  
// Column must be backed by a property on an entity.  
// Upgrader will figure out the correct datatype and nullability.  
  
table.getColumn("ColumnName").create()
```

Creating a Non-nullable Column with an Initial Value

The upgrader throws an exception if you try to add a new non-nullable column without a default value and there are rows in the table. For non-nullable columns, either specify a default value, or create a version trigger that will populate the column.

To create a new column as non-nullable with an initial value, use the `createNonNullableWithInitialValue()` method. In the data model, the column must be defined as non-nullable.

For example:

```
IBeforeUpgradeTable table = getTable("TableName")  
table.getColumn("ColumnName").createNonNullableWithInitialValue(Initial value)
```

The initial value must be of the appropriate type for the column's datatype. You can alter this value in later steps as needed.

Creating a Temporary Column

Use the `createTempColumn` method of `IBeforeUpgradeTable` to add a temporary column to the table. The `createTempColumn` method takes two parameters, a `String` for the column name and an `IDataType` for the column data type. `createTempColumn` creates a new nullable column with the given name and datatype to hold temporary data. You must explicitly drop the temporary column during the upgrade. The schema verifier will report an error during server startup if the column has not been dropped. You can create the temporary column in a `BeforeUpgradeVersionTrigger` and drop it in an `AfterUpgradeVersionTrigger`. This approach is useful when you want to move data from a column that will be removed during the upgrade to a column that will be created during the upgrade.

In the following example, a `BeforeUpgradeVersionTrigger` adds a temporary `shorttext` column to an existing entity and populates it with data from another column on a different entity. An `AfterUpgradeVersionTrigger` moves the data to a new entity.

BeforeUpgradeVersionTrigger Execute Method

```
// Add a temporary column to TableA.
var tableA = getTable("TableA")
var tempColumn = tableA.createTempColumn("tmp_column", DataTypes.shorttext())

// Get an IBeforeUpgradeUpdateBuilder for TableA.
var ub = tableA.update()

// Set the value of the temporary column to the value of ColumnA.
ub.set(tempColumn, ub.getColumnRef("ColumnA"))

ub.execute()
```

AfterUpgradeVersionTrigger Execute Method

```
// Get an IUpdateBuilder for TableA.
var ub = getTable("TableA").update().withLogSQL(true)

var q = new Query(Account).withLogSQL(true)
q.compare("ID", Equals, ub.getQuery().getColumnRef("Account"))
var piDesc = PaymentInstrument.Type.TypeInfo.getProperty("Description") as IEntityPropertyInfo

ub.set(piDesc, q, q.getColumnRef(DBFunction.Expr({"tmp_xyz"}))) // tmp_xyz is the DB table column name
ub.execute()

var tempColumn = getTable("someTable").getColumn("tmp_xyz").drop()
```

Dropping Columns

The upgrader does not drop existing columns in order to prevent data loss. You can write a version trigger to move the data (not shown in example) and then drop the column by using the `drop()` method of the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")
table.getColumn("ColumnName").drop()
```

There is a `dropColumns` method on `IBeforeUpgradeTable` to drop multiple columns in one statement. The `dropColumns` method takes an array of `IBeforeUpgradeColumn` objects.

For example:

```
var table = getTable("TableName")
table.dropColumns(table.getColumn("ColumnName2"), table.getColumn("ColumnName3"));
```

In Oracle, dropping a column usually has little effect on upgrade performance. Dropping a column actually marks the column as unused in the metadata. At a later point, the DBA is responsible for performing the necessary cleanup. You can override this functionality and force columns to be dropped right away.

In SQL Server, dropping a column is performance-intensive because the RDBMS has to do some clean up work.

Renaming Columns

To rename a column use the `rename` function on the column object.

```
override function execute() {
    getTable("TableName").getColumn("ColumnName").rename("NewColumnName")
}
```

Setting a Column Value for a Specific Subtype

To set a column to a specific value for specific subtypes, use the `set` and `compare` methods of an `IBeforeUpgradeTable`. Get the typekey ID for comparison using the `BeforeUpgradeVersionTrigger` method `getTypekeyID`.

```
final var myTable = getTable("tableName")
final var myTypecode = getTypeKeyID("typelist name", "typelist code")

final var updateBuilder = myTable.update()

updateBuilder
    .set("myColumn", "some value")
    .compare("subtype", Equals, myTypecode)
```

```
updateBuilder.execute()
```

Creating Tables

To add a new table to the database, define a new entity in the data model. The upgrade creates the table automatically. However, you might want to explicitly create the table in your upgrade trigger if you want the trigger to perform an action on the table such as populating it.

Creating a new table has negligible impact on upgrade performance.

You can create a regular table using the `create` method of `IBeforeUpgradeTable`. The table must first be defined in the data model.

For example:

```
var table = getTable("TableName").create()
```

Creating Temporary Tables

You can add a temporary table to the database based on either the current database schema for a table or the data model definition of a table. You can also create a temporary table with a custom definition.

To create a temporary table based on the current table schema in the database, use the `createNewTempTableBasedOnCurrentSchema` method of `IBeforeUpgradeTable`. The table must be associated with an entity and exist in the database. The returned temporary table will contain the columns that this table has in the database currently. The columns may not match those specified in the entity metadata. For example, the metadata might contain a new column that has not yet been created. The `createNewTempTableBasedOnCurrentSchema` method is usually more appropriate than `createNewTempTableBasedOnThis` if you want to copy data from this table into the new temporary table as the columns will match exactly.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnCurrentSchema()
```

To create a temporary table based on the entity definition of a table in the data model, use the `createNewTempTableBasedOnThis` method of `IBeforeUpgradeTable`. Columns that do not exist in the table are not created on the temporary table, even if the metadata defines such a column. This table may not contain columns that are going to be renamed. The metadata reflects the new name for the column but does not have an entry for the old name, so it would not be added to the temporary table.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnThis()
```

To create a temporary table with a custom definition, use the `createAsNewTempTable` method of `IBeforeUpgradeTable`. This method takes a `Pair` array in which the first object is a `String` defining the column name and the second object is an `IDataType` defining the column data type.

Renaming Tables

To rename a table use the `rename` function on the table object.

```
override function execute() {
    getTable("extTableName").rename("TableName_EXT")
}
```

Deleting Rows

To delete rows from a table, use the `delete` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `delete` method returns a delete builder (`IBeforeUpgradeDeleteBuilder`) that provides methods for comparing column data to restrict the rows that are deleted.

In the following example, all rows that have a columnA value of 0 are deleted.

```
var table = getTable("SomeTable")
var deleteBuilder = table.delete()
deleteBuilder.Query.compare("columnA", Equals, 0)
deleteBuilder.execute()
```

Inserting Rows

To insert rows of data use the `insert` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insert` method returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable`) for SQL to perform an insert operation.

In the following example, an `IBeforeUpgradeInsertBuilder` is used to add two rows with three columns to table `myTable`. The `IBeforeUpgradeInsertBuilder` includes a description.

```
var myTable = getTable("SomeTable")
var insertBuilder = myTable.insert().withDescription("A custom insert
trigger to add two rows.")

insertBuilder
.mapColumn("columnA", "value of column A for first row")
.mapColumn("columnB", "value of column B for first row")
.mapColumn("columnC", "value of column C for first row")

insertBuilder.execute()

// add a second row
insertBuilder
.mapColumn("columnA", "value of column A for second row")
.mapColumn("columnB", "value of column B for second row")
.mapColumn("columnC", "value of column C for second row")

insertBuilder.execute()
```

Inserting Data Selected from Another Table

To insert data selected from another table use the `insertSelect` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insertSelect` method returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`). The builder includes a `mapColumn` method that can be passed explicit values, columns, or a query.

In the following example, the trigger sets `targetTable.column1` to an explicit value. The trigger sets `targetTable.column2` to the value of `sourceTable.sourceColumn`. Because there is no comparison being performed, the trigger will insert a row in the target table for each row in the source table:

```
var sourceTable = getTable("sourceTable")
var targetTable = getTable("targetTable")

var insertSelectBuilder = targetTable.insertSelect(sourceTable)

insertSelectBuilder.mapColumn("column1", "value") // sets a hard-coded value
.mapColumn("column2", sourceTable.getColumn("sourceColumn")) // sets column2 on target table to
// source table sourceColumn

insertSelectBuilder.execute()
```

In the next example, an existing table, `sourceTable`, is split into two tables, `targetTable1` and `targetTable2`.

```
var sourceTable = getTable("sourceTable")
var targetTable1 = getTable("targetTable1")
var targetTable2 = getTable("targetTable2")

var insertSelectBuilder1 = targetTable1.insertSelect(sourceTable)
var insertSelectBuilder2 = targetTable2.insertSelect(sourceTable)

insertSelectBuilder1.mapColumn("column1", sourceTable.getColumn("sourceColumn1"))
.mapColumn("column2", sourceTable.getColumn("sourceColumn2"))

insertSelectBuilder1.execute()
```

```
insertSelectBuilder2.mapColumn("column1", sourceTable.getColumn("sourceColumn3"))
    .mapColumn("column2", sourceTable.getColumn("sourceColumn4"))

insertSelectBuilder2.execute()
```

Updating Rows

To update rows in a table, use the update method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. This method returns a builder (`IBeforeUpgradeUpdateBuilder` or `IUpdateBuilder`). The builder includes methods to compare data to restrict which rows are updated.

In the following example, table `SomeTable` is updated to set `column1` to `SomeValue` for each row where the subtype matches a certain entity type:

```
var table = getTable("SomeTable")

// get IBeforeUpgradeUpdateBuilder
var ub = table.update()

// set column 1 to SomeValue
ub.set("column1", "SomeValue")
// where
    .compare("subType", Equals, getTypeKeyID(EntityType))
    .execute()
```

Running the Commission Payable Calculations Process

Run the Commission Payable Calculations process on your starting version before you upgrade the database. The Commission Payable Calculations process makes commission payable on direct bill policies.

To run the Commission Payable Calculations process

1. Start your pre-upgrade BillingCenter server.
2. Open BillingCenter and log in with an administrator account.
3. Open Server Tools by pressing ALT + SHIFT + T.
4. Click Batch Process Info.
5. In the Action column for Commission Payable Calculations, click Run. Wait for the process to complete before upgrading BillingCenter.

For more information about the Commission Payable Calculations process, see “Commission Payable Calculations Batch Processing” on page 126 in the *System Administration Guide*.

Configuring the Database Upgrade

You can set parameters for the database upgrade in the `BillingCenter 8.0.4 database-config.xml` file. The `<database>` block in `database-config.xml` contains parameters for database configuration, such as connection information. The `<database>` block contains an `<upgrade>` block that contains configuration information for the overall database upgrade. The `<upgrade>` block also contains a `<versiontriggers>` element for configuring general version trigger behavior and can contain `<versiontrigger>` elements to configure each version trigger.

This topic describes the parameters you can set for the database upgrade. For general database connection parameters, see “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.

Adjusting Commit Size for Encryption

You can adjust the commit size for rows requiring encryption by setting the `encryptioncommitsize` attribute to an integer in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade encryptioncommitsize="10000">
    ...
  </upgrade>
</database>
```

If BillingCenter encryption is applied on one or more attributes, the BillingCenter database upgrade commits batches of encrypted values. The upgrade commits `encryptioncommitsize` rows at a time in each batch. The default value of `encryptioncommitsize` varies based on the database type. For Oracle, the default is 10000. For SQL Server, the default is 100.

Test the upgrade on a copy of your production database before attempting to upgrade the actual production database. If the encryption process is slow, and you cannot attribute the slowness to SQL statements in the database, try adjusting the `encryptioncommitsize` attribute. After you have optimized performance of the encryption process, use that `encryptioncommitsize` when you upgrade your production database.

Configuring Version Trigger Elements

The database upgrade executes a series of version triggers that make changes to the database to upgrade between versions. You can set some configuration options for version triggers in `database-config.xml`. Normally, the default settings are sufficient. Change these settings only while investigating a slow database upgrade.

The `<database>` element in `database-config.xml` contains an `<upgrade>` element to organize parameters related to database upgrades. Included in the `<upgrade>` element is a `<versiontriggers>` element, as shown below:

```
<database ...>
  <param ... />
  <upgrade>
    <versiontriggers dbmsperfinfotreshold="600" />
  </upgrade>
</database>
```

The `<versiontriggers>` element configures the instrumentation of version triggers. This element has one attribute: `dbmsperfinfotreshold`. The `dbmsperfinfotreshold` attribute specifies for all version triggers the threshold after which the database upgrader gathers performance information from the database. You specify `dbmsperfinfotreshold` in seconds, with a default of 600. If a version trigger takes longer than `dbmsperfinfotreshold` to execute, BillingCenter:

- queries the underlying database management system (DBMS).
- builds a set of html pages with performance information for the interval in which the version trigger was executing.
- includes those html pages in the upgrader instrumentation for the version trigger.

You can completely turn off the collection of database snapshot instrumentation for version triggers by setting the `dbmsperfinfotreshold` to 0 in `config.xml`.

The `<versiontriggers>` element can contain optional `<versiontrigger>` elements for each version trigger. Each `<versiontrigger>` element can contain the following attributes.

Attribute	Type	Description
<code>name</code>	String	The case-insensitive name of a version trigger.
<code>extendedquerytracingenabled</code>	Boolean	Oracle only. Controls whether or not to enable extended sql tracing (Oracle event 10046) for the SQL statements that are executed by the version trigger. Default is <code>false</code> . The output can be very useful when debugging certain types of performance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation.
<code>parallel-dml</code>	Boolean	Oracle only. See “Configuring Parallel DML and DDL Statement Execution” on page 74.
<code>queryoptimizertracingenabled</code>	Boolean	Oracle only. Controls whether or not to enable query optimizer tracing (Oracle event 10053) for the SQL statements that are executed by the version trigger. Default is <code>false</code> . The output can be very useful when debugging certain types of performance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation.
<code>recordcounters</code>	Boolean	Controls whether the DBMS-specific counters are retrieved at the beginning and end of the use of the version trigger. Default is <code>false</code> . If true, then BillingCenter retrieves the current state of the counters from the underlying DBMS at the beginning of execution of the version trigger. If the execution of the version trigger exceeds the <code>dbmsperfinfotreshold</code> , then BillingCenter retrieves the current state of the counters at the end of the execution of the version trigger. BillingCenter writes differences to the DBMS-specific instrumentation pages of the upgrade instrumentation.
<code>updatejoinorderedhint</code>	Boolean	Oracle only. Whether to use the ORDERED hint for the UPDATE of a join. Default is <code>false</code> .
<code>updatejoinusemergehint</code>	Boolean	Oracle only. Whether to use the USE_MERGE hint for the UPDATE of a join. Default is <code>false</code> .
<code>updatejoinusenlhint</code>	Boolean	Oracle only. Whether to use the USE_NL hint for the UPDATE of a join. Default is <code>false</code> .

Deferring Creation of Nonessential Indexes

You can configure the upgrade to defer creation of nonessential indexes during the upgrade process until the upgrade completes and the application server is online. Nonessential indexes are performance-related indexes that do not enforce constraints. Creation of nonessential indexes can add significant time to the upgrade duration, so it is possible to defer this process. By default, the upgrade does not defer creation of these indexes.

To configure the upgrade to defer creation of nonessential indexes set the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` to `true`.

```
<database ...>
  <upgrade defer-create-nonessential-indexes="true">
    ...
  </upgrade>
</database>
```

If you opt to defer creation of nonessential indexes, BillingCenter runs the `DeferredUpgradeTasks` batch process as soon as the upgrade completes and the server is completely started. The `DeferredUpgradeTasks` batch process creates the nonessential performance indexes. The database user must have permission to create indexes until after the `DeferredUpgradeTasks` batch process is complete.

Deferring nonessential index creation can shorten the duration of the upgrade process. The BillingCenter database is then available sooner for tasks including upgrade verification and backing up the upgraded database before the database is opened up for production use. To take advantage of this earlier availability, perform upgrade testing and validation tasks while the `DeferredUpgradeTasks` batch process is running. Do not go into full production while the process is still running. The lack of so many performance-related indexes could likely make the system unusable.

Until the `DeferredUpgradeTasks` batch process has run to completion, BillingCenter reports errors during schema validation when starting. These include errors for column-based indexes existing in the data model but not in the physical database and mismatches between the data model and system tables.

Check the status of the `DeferredUpgradeTasks` batch process to determine when it has completed successfully. You can find the status of the deferred upgrade in the upgrade logs and on the BillingCenter [Upgrade Info](#) page. If the `DeferredUpgradeTasks` batch process fails, manually run the batch process again during non-peak hours.

If you do not opt to defer creation of nonessential indexes, BillingCenter creates these indexes as part of the upgrade process that must complete before the application server is online. If you do not want to defer creating nonessential indexes, the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` must be set to `false`. This is the default setting.

Configuring the Upgrade on Oracle

Configuring Column Removal

The database upgrade removes some columns. For Oracle, you can configure whether the removed columns are dropped immediately or are marked as unused. Marking a column as unused is a faster operation than dropping the column immediately. However, because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments. You can drop the unused columns after the upgrade during off-peak hours to free the space. Or, you can configure the database upgrade to drop the columns immediately during the upgrade. By default, the BillingCenter database upgrade marks columns as unused.

To configure the BillingCenter upgrade to drop columns immediately during the upgrade, set the `deferDropColumns` attribute of the `<upgrade>` block in `database-config.xml` to `false`. For example:

```
<database ...>
  ...
  <upgrade deferDropColumns="false">
    ...
  </upgrade>
</database>
```

By default, `deferDropColumns` is `true`.

Configuring Parallel DML and DDL Statement Execution

You can configure whether the upgrade executes DML (Data Manipulation Language) and DDL (Data Definition Language) statements in parallel or not and the degree of parallelism to use.

The `<upgrade>` element includes an `ora-parallel-dml` attribute. This attribute can be set to `disable`, `enable`, or `enable-all`. The default value is `enable`. If `ora-parallel-dml` is set to `disable`, the upgrade does not conduct parallel execution of DML statements. If `ora-parallel-dml` is set to `enable`, the upgrade executes DML statements in parallel if configured or coded for a version trigger. If `ora-parallel-dml` is set to `enable-all`, the upgrade executes DML statements in parallel in all cases unless turned off in the code or configuration for a version trigger.

The Boolean attribute `parallel-dml` of a `<versiontrigger>` element controls parallel execution for that version trigger. If `parallel-dml` is not set, the upgrade executes parallel DML statements if coded or if `ora-parallel-dml` is set to `enable_all` on the `<upgrade>` element. If `parallel-dml` is set to `false`, the upgrade does not execute DML statements in parallel. If `parallel-dml` is set to `true`, the upgrade executes DML statements in parallel if `ora-parallel` is set to `enable` or `enable_all`.

To configure the degree of parallelism for insert, update and delete operations, set the `degree-of-parallelism` attribute on the `<upgrade>` element. To configure the degree of parallelism for commands such as creating an index and enabling constraints using the alter table command, set the `degree-parallel-ddl` attribute on the `<upgrade>` element.

You can specify a value from 2 to 1000 to force that degree of parallelism. Specify a value of 1 to disable the use of parallel execution.

Setting either parameter to 0 configures BillingCenter to defer to Oracle to determine the degree of parallelism for the operations that attribute configures. The Oracle automatic parallel tuning feature determines the degree based on the number of CPUs and the value set for the Oracle parameter `PARALLEL_THREADS_PER_CPU`.

The default for both attributes is 4.

You can configure parallel DML execution on the `InsertSelectBuilder`, `BeforeUpgradeUpdateBuilder` and `BeforeUpgradeInsertSelectBuilder` of a custom version trigger using the `withParallelDml(boolean)` method. If not explicitly set to `true` or `false`, the upgrade uses parallel execution if configured. If set to `false`, the upgrade does not use parallel execution unless set to `true` for that version trigger. If set to `true`, it will be done unless set to `false` for that version trigger or `ora-parallel-dml` is set to `disable`.

Collecting Tablespace Usage and Object Size

To enable collection of tablespace usage and object size data on Oracle, set the `collectstorageinstrumentation` attribute of the `<upgrade>` block to `true`. For example:

```
<database ...>
...
<upgrade collectstorageinstrumentation="true">
...
</upgrade>
</database>
```

A value of `true` enables BillingCenter to collect tablespace usage and size of segments such as tables, indexes and LOBs (large object binaries) before and after the upgrade. The values can then be compared to find the utilization change caused by the upgrade.

Disabling Oracle Logging

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
...
<upgrade allowUnloggedOperations="true">
...
</upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run statements with the `NOLOGGING` option.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. Some examples include Reporting, Disaster Recovery through Standby databases and Oracle Dataguard. To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

Disabling Statistics Update for the Database

Generating table statistics during upgrade is optional for Oracle databases. The overall time required to upgrade the database is shorter if the database upgrade does not update statistics. To disable statistics generation during the upgrade, set the `updatestatistics` attribute of the `<upgrade>` element to `false`:

```
<upgrade updatestatistics="false">
```

If `updatestatistics` is `true`, the upgrade updates statistics and deletes histograms on columns for which BillingCenter does not generate statistics. This setting enables the upgrade to update statistics on changed objects. It also configures BillingCenter to maintain column-level statistics consistent with what is allowed in the code, data model and configuration.

If statistics are not updated during the upgrade, BillingCenter reports a warning that recommends that you run the database statistics batch process in incremental mode. Additionally, the [Upgrade Info](#) page shows that statistics were not updated as part of the upgrade. If statistics generation was not disabled, the [Upgrade Info](#) page reports the runs of the statistics batch process, including incremental runs.

You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded BillingCenter in a production environment. If you defer generating statistics during the upgrade, Guidewire recommends that you generate full statistics as soon as possible after the upgrade. For instructions, see “Commands for Updating Database Statistics” on page 40 in the *System Administration Guide*.

The [Upgrade Info](#) page does not identify the following case: You ran an upgrade with `updatestatistics=true` after running a previous upgrade with `updatestatistics=false`, but you did not update statistics first.

When you click the **Download** button on the [Upgrade Info](#) page, you get a more detailed report. This report shows the value of the `updatestatistics` attribute at the time of upgrade. Additionally, the report shows the update statistics SQL statements that were skipped as part of the upgrade. These statements are provided for reference. You typically do not need to review these statements if you run the incremental database statistics process following the upgrade.

Disabling Statistics Update for Tables with Locked Statistics

If you have tables that have locked statistics, specify to keep statistics on these tables before starting the database upgrade. To specify to keep statistics on a table, set the `action` attribute of the `<tablestatistics>` element for that table to `keep`. The `<tablestatistics>` element is nested within the `<databasestatistics>` element, which is within the `<database>` element in `database-config.xml`.

For example, if statistics are locked on `bc_someTable_EXT`, specify a `<tablestatistics>` element for that table with the `action` attribute set to `keep`:

```
<database>
  ...
  <databasestatistics>
    <tablestatistics name="bc_someTable_EXT" action="keep" />
  </databasestatistics>
</database>
```

Configuring the Upgrade on SQL Server

Disabling SQL Server Logging

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="true">
    ...
  </upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run with minimal logging. This can improve the performance of the upgrade. During the upgrade, set the SQL Server recovery model to Simple or Bulk logged. Once the upgrade and deferred upgrade tasks are complete, you can revert the recovery model setting and back up the full database.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. If you require full logging due to the presence of solutions such as Database Mirroring, continue to use the Full recovery model and set `allowUnloggedOperations` to `false`.

To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

Storing Temporary Sort Results in tempdb

For SQL Server databases, you can specify to store temporary sort results in tempdb by setting the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` block to `true`. By using tempdb for sort runs, disk input and output is typically faster, and the created indexes tend to be more contiguous. By default, `sqlserverCreateIndexSortInTempDB` is `false` and sort runs are stored in the destination filegroup.

If you set `sqlserverCreateIndexSortInTempDB` to `true`, you must have enough disk space available to tempdb for the sort runs, which for the clustered index include the data pages. You must also have sufficient free space in the destination filegroup to store the final index structure, because the new index is created before the old index is deleted. Refer to <http://msdn.microsoft.com/en-us/library/ms188281.aspx> for details on the requirements to use tempdb for sort results.

Specifying Filegroup to Store Sort Results for Clustered Indexes

For SQL Server databases, a version trigger recreates non-clustered backing indexes for primary keys as clustered indexes.

Before recreating the indexes, the version trigger automatically drops (and later rebuilds) any referencing foreign keys and drops any clustered indexes on tables with a primary key.

If you are using filegroups, the upgrade recreates the clustered index in the OP filegroup. By default, the upgrade also stores the intermediate sort results that are used to build the index in the OP filegroup. You can configure the upgrade to instead use the tempdb filegroup for the intermediate sort results.

If you want the upgrade to stores the intermediate sort results in the tempdb filegroup, set the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` element to `true`.

```
<database ...>
  ...
  <upgrade sqlserverCreateIndexSortInTempDB="true" />
  ...
</upgrade>
</database>
```

This option increases the amount of temporary disk space that is used to create an index. However, it might reduce the time that is required to create or rebuild an index when tempdb is on a different set of disks from that of the user database.

By default, `sqlserverCreateIndexSortInTempDB` is `false`.

Downloading Database Upgrade Instrumentation Details

The database upgrade deletes upgrade instrumentation information for prior database upgrades. If the database upgrade detects any prior upgrade instrumentation data, it reports a warning and deletes the data. If you have run previous database upgrades, and you want to preserve upgrade instrumentation details, follow the procedure in “Viewing Detailed Database Upgrade Information” on page 83.

Checking the Database Before Upgrade

The upgrade runs a series of version checks prior to making any changes to the database. These version checks ensure that the database is in a state that can be upgraded. Guidewire includes a number of version checks with BillingCenter and you can also add custom version checks.

You can configure BillingCenter to run the version checks only, including custom version checks. Before upgrading the production database, run version checks on a clone of your production database to identify any issues with your data.

To run version checks without database upgrade

1. Start Studio for BillingCenter 8.0.4 by running the following command from the bin directory:
`gwbc studio`
2. Expand **configuration** → **config** and open **database-config.xml**.
3. Add the attribute `versionchecksonly=true` to the `database` element. The `versionchecksonly` attribute overrides the `autoupgrade` attribute. If both are set to true, BillingCenter only runs version checks when the server starts.
4. Verify that the database connection is pointing to a clone of your production database.
5. Save your changes.
6. Start the server.

BillingCenter reports the number of version check errors. For any errors reported BillingCenter reports which version check resulted in the error along with the error message.

If BillingCenter reports version check errors, fix the data and rerun the version checks. Repeat this process until no errors are reported on the production clone. Apply the fixes to your production database prior to upgrade.

With `versionchecksonly=true` set, BillingCenter runs all version checks regardless of a failure in one of the checks. During a regular upgrade, BillingCenter stops the upgrade if an error is detected.

After you have fixed all version check errors, set `versionchecksonly` to `false` to run the actual upgrade.

Disabling the Scheduler

Before you start the server to upgrade the database, disable the scheduler for batch processes and work queues. Disabling the scheduler prevents batch processes and work queues from launching immediately after the database upgrade.

To disable the scheduler

1. Open the BillingCenter 8.0.4 `config.xml` file in a text editor.
2. Set the `SchedulerEnabled` parameter to `false`.
`<param name="SchedulerEnabled" value="false"/>`
3. Save `config.xml`.

After you have successfully upgraded the database, you can enable the scheduler by setting `SchedulerEnabled` to `true`. This can be accomplished by performing the database upgrade using a WAR or EAR file that has the `SchedulerEnabled` parameter to `false`. After the upgrade is complete and verified, stop the server and deploy a new WAR or EAR file that differs from the first only by having `SchedulerEnabled` set to `true`. Finally, restart the server to activate the scheduler.

Suspending Message Destinations

Suspend all event message destinations before you upgrade the database to prevent BillingCenter from sending messages until you have verified a successful database upgrade.

To suspend message destinations

1. Start the BillingCenter server for the pre-upgrade version.
2. Log in to BillingCenter with an account that has administrative privileges, such as the superuser account.
3. Click the **Administration** tab.
4. Click **Event Messages**.
5. Select the check box to the left of the **Destination** column to select all message destinations.
6. Click **Suspend**.

Resume messaging after you have verified a successful database upgrade.

Starting the Server to Begin Automatic Database Upgrade

The database upgrade is an automatic process that occurs as you start the server with the upgraded configuration of a new BillingCenter version. The database upgrade normally completes in a few hours or less.

If the database upgrade stops before completing, then restore your database from the backup, correct any issues reported, and repeat the database upgrade.

IMPORTANT Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

WARNING Except for your first database upgrade trials, do not start the server until you have upgraded all rules. Otherwise, default validation rules execute. This could strand objects at a high validation level and make it impossible to edit parts of the object.

WARNING The database upgrade runs a series of version checks prior to making any changes. If any of these checks fail, the upgrade aborts and reports an error message. You can fix the issue, create an updated backup of the database and attempt the upgrade again without restoring from a backup. However, if you experience a failure during the version triggers or upgrade steps portion of the upgrade, refresh the database from a backup before attempting the upgrade again.

Test the Database Upgrade

Prior to attempting the database upgrade on a full-production database clone, test the database upgrade.

To test the database upgrade

1. Connected to the built-in Quickstart database, successfully start the built-in Quickstart application server with a merged configuration data model, including merged extensions, data types, field validators, and so forth.

2. Connected to an empty database on an Oracle or SQL Server database server, successfully start the Quickstart application server from the preceding step.
3. Connected to a restored backup of a production clone, start either the same Quickstart server from the preceding step or a supported third-party application server with your custom configuration.

In a development environment the database upgrade process records checkpoints of upgrade triggers that complete successfully. You can restart a failed database upgrade, and it resumes with the upgrade trigger that failed. This restart feature helps you test the upgrade with realistically large data sets. You avoid time spent to restore the database and rerun upgrade triggers that worked successfully.

Guidewire provides this feature for convenience while testing. However, it does not work for all failure scenarios. Even in development mode, under certain scenarios, you will have to restore a backup of the database taken prior to the upgrade attempts and then run the upgrade.

The database upgrade writes SQL executed by the failed trigger to the console. To restart a test database upgrade from a checkpoint reached in an earlier upgrade, manually roll back any database changes that occurred during the upgrade trigger that failed. Resolve the problem that caused the trigger to fail. Then start the server again to restart the upgrade. The upgrade skips successful upgrade triggers and continues by rerunning the trigger that failed.

A test run of your upgrade is successful only when it runs from start to finish without a restart.

WARNING Never use the restart feature of database upgrade in a production environment.

Integrations and Starting the Server

Disable all integrations during the automatic database upgrade. Integration points might require updates due to changes in Guidewire APIs. See the *BillingCenter New and Changed Guide* for specifics.

It is not necessary to have completely migrated integrations before attempting to start the server for the first time. If you have integrations that rely on non-Guidewire applications, do not expect these integrations to work the first time you start the server.

Understanding the Automatic Database Upgrade

As the database upgrade proceeds, it logs messages to the console as well as the log file describing its progress. The database upgrade process requires thousands of steps, divided into three phases. Due to the relational nature of a database, these phases must execute in a specific order for the upgrade to succeed.

During the first phase, the upgrader first executes custom `BeforeUpgradeVersionTrigger` version checks and triggers defined in the `IDataModelUpgrade` plugin. The upgrader next runs version checks defined by Guidewire. Then, the upgrader uses a set of version triggers defined by Guidewire to determine the actions that are required. The database upgrader requires version triggers in order to perform the following types of tasks:

- changing a datatype (other than just length)
- migrating data
- dropping a column
- dropping a table
- renaming a column
- renaming a table

Specific version triggers are described in this topic.

Many version triggers have version checks associated with them. These checks ensure that the database is ready for the associated version trigger. The database upgrade runs all checks before running any version triggers. If a check detects a problem, it reports the issue, including a sample SQL query to find specific problematic records. If a version check discovers an issue, the database upgrade stops before any version triggers are run. Therefore, it is not necessary to restore the database from a backup if a version check reports an error. Correct the issue and then create a new backup of the database. Then, if you encounter errors after the version check stage, you can restore a version of your database with the issue reported by the version check resolved.

In the second phase, the upgrader compares the target data model and the current database to determine how they differ. The upgrader makes changes to the database that do not require a version trigger during this phase.

Following this process, the third phase runs a subsequent set of version triggers. These triggers create actions that must be run last due to a dependency on an earlier phase.

After the database upgrade concludes, it reports issues that the upgrader encountered and did not complete.

You are responsible for correcting these issues. This might involve modifying the data model or altering the table manually. If you do not correct them, the next time you start the server you do *not* see a message that the database and the data model are out of sync. You must then use the `system_tools` command to verify the database schema.

Note: Given the complexity of database upgrade, Guidewire does not expose specific upgrade actions/steps to clients either in SQL or Java form. Any manual attempts to recreate or control the upgrade process can result in problems in the BillingCenter database. Recovery from such attempts is not supported.

Version Trigger Descriptions

The database upgrade uses version triggers to perform the actions described by sections within this topic. If a version trigger has an associated version check, the check is described with the trigger. Review these descriptions to familiarize yourself with some of the changes and to understand version checks. If a version check reports an issue, review the error message and consult the description of the relevant version trigger for more information.

Renaming Deferred Upgrade Batch Process

The upgrade renames the `DeferredUpgrade` batch process type to `DeferredUpgradeTasks`.

Truncating bc_Dynamic_Assign

The upgrade truncates the `bc_Dynamic_Assign` table.

Dropping bc_t1_Template

The upgrade drops the `bc_t1_Template` table.

Dropping Columns from WorkItem Tables

The upgrade drops the `AvailableSince` and `LastUpdateTime` columns from all `bc_WorkItem` tables.

Upgrading Shared Typekey Data

The upgrade checks for subtypes with typekeys that have the same field name, different column names, and only one column exists in the database. If any such records exist, the upgrade moves the data to the correct column.

Renaming LOBCode Typekey

The upgrade renames the `LOBCode` typekey `HomeOwners` to `Homeowners`.

Dropping DunningInterval from Plan

The upgrade drops the `bc_Plan.DunningInterval` column.

Adding ListBillAccountExcessTreatment to ReturnPremiumPlan

The upgrade adds a `ListBillAccountExcessTreatment` column to all `ReturnPremiumPlan` instances with value of `POLICY_PAYER_UNAPPLIED`.

Adding PaymentAllocationPlans

The upgrade adds `PaymentAllocationPlans` representing each distribution limit and links to each `Account` according to the previous `DistributionLimitType` of the `Account`.

Updating Denormalized Fields on InvoiceItem

The upgrade updates denormalized fields `CanBePaidMoreByAgencyBill` and `CanBePromisedMoreByAgencyBill` on `bc_InvoiceItem` to indicate whether or not each invoice item can be paid or promised any more with agency billing.

Associating Users with User-specific Custom Authority Limit Profiles

The upgrade adds records to `bc_CustomALPUser` to associate users with user-specific custom authority limit profiles and deletes `bc_AuthorityLimitProfile.Custom` if it exists.

Creating and Updating AccountContext.UnappliedFundID

The upgrade creates and updates the `bc_AccountContext.UnappliedFundID` foreign key. For `AccountContext` records, the upgrade sets the `UnappliedFundID` to the `DefaultUnappliedFund` of the associated `Account`. For `DisbPaidContext` records, the upgrade sets the `UnappliedFundID` to the `UnappliedFund` of the associated `AccountDisbursement`. For `AccountNegativeWriteoffContext` records, the upgrade sets the `UnappliedFundID` to the `UnappliedFund` of the associated `AcctNegativeWriteoff`.

Adding Payment Allocation Privileges Included with BillingCenter 8.0.1

The upgrade adds the following payment allocation privileges:

Privilege	Roles
<code>payallocplancreate</code>	<code>general_admin</code> <code>plan_admin</code> <code>superuser</code>
<code>payallocplanedit</code>	<code>general_admin</code> <code>plan_admin</code> <code>superuser</code>
<code>payallocplanview</code>	<code>billing_clerical</code> <code>billing_manager</code> <code>commissions_admin</code> <code>finance_manager</code> <code>general_admin</code> <code>plan_admin</code> <code>superuser</code> <code>underwriter</code>

Adding Direct Collector Role

The upgrade adds the Direct Collector role.

Dropping PolicyDInqProcessID from LegacyDInqWorkItem

The upgrade drops the `bc_LegacyDInqWorkItem.PolicyDInqProcessID` column.

Viewing Detailed Database Upgrade Information

BillingCenter includes an **Upgrade Info** page that provides detailed information about the database upgrade. The **Upgrade Info** page includes information on the following:

- version numbers before and after the database upgrade
- configuration parameters used during the database upgrade
- SQL queries for version checks that test if the database is in condition to be upgraded
- changes made to specific tables, including which version triggers modified the table or its data and the SQL statement executed to make each change
- version triggers that the upgrade ran, including which tables the trigger ran against, a description, the SQL statement run against each table and the start and end time
- a list of upgrade steps, including the table on which the step operated
- a table registry including table IDs before and after upgrade

The database upgrade deletes upgrade instrumentation information for prior database upgrades. If the database upgrade detects any prior upgrade instrumentation data, it reports a warning and deletes the data. If you have run previous database upgrades, and you want to preserve upgrade instrumentation details, download this information.

To download upgrade instrumentation details

1. Start the BillingCenter server if it is not already running.
2. Log in to BillingCenter with the superuser account.
3. Press ALT+SHIFT+T to access **System Tools**.
4. Click **Info Pages**.
5. Select **Upgrade Info** from the **Info Pages** drop-down.
6. Click **Download** to download a ZIP file containing the detailed upgrade information.

Dropping Unused Columns on Oracle

By default, the BillingCenter database upgrade on Oracle marks columns that have been removed from the data model as unused. Marking a column unused is a faster operation than dropping a column. Because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments.

You can configure the upgrade to drop removed columns immediately by setting the `deferDropColumns` parameter to `false` before running the database upgrade. This parameter is within the `<upgrade>` block of the `<database>` block of `database-config.xml`.

If you did not set `deferDropColumns` to `true` before the upgrade, perform the procedure in this topic to drop unused columns after the upgrade. You can drop the unused columns after the upgrade during off-peak hours to free the space. BillingCenter does not have to be shutdown to perform this maintenance task. You can drop all unused columns in one procedure, or you can drop unused columns for individual tables.

To drop all unused columns

1. Create the following Oracle procedure to purge all unused columns:

```
DECLARE
    dropstr VARCHAR2(100);
    CURSOR unusedcol IS
        SELECT table_name
        FROM user_unused_col_tabs;
BEGIN
    FOR tabs IN unusedcol LOOP
        dropstr := 'alter table '
                   || tabs.table_name
                   ||' drop unused columns';
        EXECUTE IMMEDIATE dropstr;
    END LOOP;
END;
```

2. Run the procedure during a period of relatively low activity.

To drop unused columns for a single table (or all tables)

1. Start the server to run the schema verifier. The schema verifier runs each time the server starts. If there are unused columns, the schema verifier reports a difference between the physical database and the data model. The schema verifier reports the name of each table and provides an SQL command to remove unused columns from each table.
2. Run the SQL command provided by the schema verifier. This command has the following format:

```
ALTER TABLE tableName DROP UNUSED COLUMNS
```

Exporting Administration Data for Testing

Guidewire recommends that you create a small set of administration data from an upgraded data set. Use this data for development and testing of rules and libraries with BillingCenter 8.0.4. This procedure is optional.

You might have already created an upgraded administration data set by following the procedure “Upgrading Administration Data for Testing” on page 52. If you followed that procedure, or you do not want an administration-only data set for testing purposes, you can skip this topic.

To create an administration data set for testing

1. Export administration data from your upgraded production database.

- a. Start the BillingCenter 8.0.4 server by navigating to BillingCenter/bin and running the following command:

```
gwbc dev-start
```

- b. Open a browser to BillingCenter 8.0.4.

- c. Log on as a user with the viewadmin and soapadmin permissions.

- d. Click the **Administration** tab.

- e. Click → Utilities → Export Data.

- f. Select the **Admin** data set to export.

- g. Click **Export** to download the **admin.xml** file.

2. Create a new database account for the development environment on a database management system supported by BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <http://guidewire.custhelp.com>.

See “Configuring the Database” on page 23 in the *Installation Guide* for instructions to configure the database account.

3. Install a new BillingCenter 8.0.4 development environment. Connect this development environment to the new database account that you created in step 2. See the *BillingCenter Installation Guide* for instructions.
4. Copy the `admin.xml` file that you exported to a location accessible from the new development environment.
5. Create an empty version of `importfiles.txt` in the `modules/configuration/config/import/gen` directory of the new development environment.
6. Create empty versions of the following CSV files:
 - `activity-patterns.csv`
 - `authority-limits.csv`
 - `reportgroups.csv`
 - `roleprivileges.csv`
 - `rolereportprivileges.csv`Leave `roles.csv` as the original complete file.
7. Import the administration data into the new database:
 - a. Start the BillingCenter 8.0.4 development server by navigating to `BillingCenter/bin` and running the following command:
`gwbc dev-start`
 - b. Open a browser to BillingCenter 8.0.4.
 - c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
 - d. Click the **Administration** tab.
 - e. Click → Utilities → Import Data.
 - f. Click **Browse....**
 - g. Select the `admin.xml` file that you exported from the upgraded production database and modified.
 - h. Click **Open**.

Final Steps After The Database Upgrade is Complete

This section describes the procedures to run after you have completed the upgrade procedure and migration of configurations and integrations. The processes in this section provide you with a benchmark of the new system. Completing these steps is particularly important to going live in a production environment.

Use these procedures to revalidate the database:

- “Validating the Database Schema” on page 55
- “Checking Database Consistency” on page 56, including “Checking that Contacts Have Unique Addresses” on page 86
- “Creating a Data Distribution Report” on page 56
- “Generating Database Statistics” on page 57. You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded BillingCenter in a production environment.
- “Backing up the Database After Upgrade” on page 86

Checking that Contacts Have Unique Addresses

An Address cannot be shared by more than one Contact. BillingCenter 8.0 includes a commit-time check that does not allow a shared reference to an address instance even when one of the referring Contact or ContactAddress instances is retired. If you have multiple contacts at the same address, you can create separate address instances with the same field values.

A database consistency check on the Contact entity reports an error if it detects multiple Contact records using the same PrimaryAddress.

Before using BillingCenter 8.0.4 in production, run database consistency checks to find any instances of shared references to address instances. If the consistency check reports shared addresses, contact Guidewire Support for assistance fixing your database.

Completing Deferred Upgrade

If you have archiving enabled, and you did not set deferCreateArchiveIndexes to false, run the Deferred Upgrade Tasks batch process as soon as possible after the completion of the upgrade. To run the Deferred Upgrade Tasks batch process, use the admin/bin/maintenance_tools command:

```
maintenance_tools -password password -startprocess deferredupgradetasks
```

Reenabling Database Logging

You might have disabled logging of direct insert and create index operations during the database upgrade. After you complete the database upgrade successfully, you can reenable logging by setting allowUnloggedOperations to false in the <upgrade> block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="false">
    ...
  </upgrade>
</database>
```

For SQL Server, if you changed the recovery model from Full to Simple or Bulk logged during the upgrade, you can revert the recovery model. If you deferred migrating to 64-bit IDs, you might disable logging again when you perform the migration.

Backing up the Database After Upgrade

Finally, before going live, back up the upgraded database. This provides you with a snapshot of the initial upgraded data set, if an unanticipated event occurs just after going live.

Upgrading BillingCenter from 8.0.x for ContactManager

This topic lists the manual tasks required to upgrade BillingCenter 8.0.x and ContactManager 8.0.x to BillingCenter 8.0.4 and ContactManager 8.0.4. Before starting this upgrade process, you must have run the Guidewire upgrade and merge tools. Additionally, Guidewire recommends that you first upgrade ContactManager, integrate BillingCenter with ContactManager, and refresh the ContactManager web APIs.

This topic includes:

- “Manually Upgrading BillingCenter to Integrate with ContactManager” on page 87
- “File Changes in BillingCenter Related to ContactManager” on page 88

Manually Upgrading BillingCenter to Integrate with ContactManager

This topic describes tasks you might have to perform to complete a BillingCenter 8.0.x upgrade when you have ContactCenter installed.

Prior to performing the tasks in this topic, do the following:

1. Run the Configuration Upgrade tool and perform the automatic upgrade for the BillingCenter configuration. See “Upgrading the BillingCenter 8.0.x Configuration” on page 29. Do not make changes yet to the files listed in this topic for BillingCenter. You make those changes later as described in this topic.
2. Run the Database Upgrade tool to upgrade for the BillingCenter database. See “Upgrading the BillingCenter 8.0.x Database” on page 51.
3. You can perform any manual configuration upgrades except those related to files listed later in this topic. Before making those changes, wait until you configure ContactManager, regenerate its SOAP API, and refresh that API in BillingCenter Studio, as described in the topics that follow.
4. Manually configure ContactManager. See “Upgrading ContactManager from 8.0.x” on page 89.

5. Integrate BillingCenter and ContactManager as described at “Integrating ContactManager with Guidewire Core Applications” on page 45 in the *Contact Management Guide*.

File Changes in BillingCenter Related to ContactManager

Web Service Version Changes

The following BillingCenter web services changed in 8.0.1 and now have folders or names that use bc801 instead of bc800 or ab801 instead of ab800:

- BillingCenter now uses `wsi.remote.gw.webservice.ab.ab801.wsc` to access the ContactManager web service `${ab}/ws/gw/webservice/ab/ab801/abcontactapi/ABContactAPI?wsdl`. See “Step 1: Integrate ContactManager with BillingCenter” on page 61 in the *Contact Management Guide*.
- BillingCenter has moved all its 8.0.1 web services to directories with a bc801 folder. For example,
 - The BillingCenter implementation of ABClientAPI, `ContactAPI.gs`, is now in the package `gw.webservice.bc.bc801.contact`. See “Synchronizing BillingCenter and ContactManager Contacts” on page 188 in the *Contact Management Guide*.
 - The `ContactAPI.wsdl` file is accessible in the BillingCenter Studio Project window in `configuration → gsrc` at `wsi.local.gw.webservice.bc.bc801.contact`.

Upgrading ContactManager from 8.0.x

This topic covers the manual steps needed to perform an upgrade of ContactManager 8.0.x to ContactManager 8.0.4. Prior to performing these upgrade steps, you must run the upgrade software and perform automatic upgrades.

This topic includes:

- “Manually Upgrading the ContactManager Configuration” on page 89

Manually Upgrading the ContactManager Configuration

Because ContactManager has changed a number of the files used to integrate with the Guidewire applications, it is likely that you will need to manually update configuration files. In particular, you will need to make manual updates:

- If you have made changes to the ABContact data model.
- If you have changed any of the files that are listed in “Manually Configuring Changed Files” on page 90.

In general, the steps for upgrading ContactManager are:

1. Run the configuration upgrade tool and perform an automatic upgrade of the ContactManager configuration. See “Upgrading the BillingCenter 8.0.x Configuration” on page 29.
2. Run the database upgrade tool to upgrade the ContactManager database. See “Upgrading the BillingCenter 8.0.x Database” on page 51.
3. Manually configure files in ContactManager—the subject of this topic.
4. Upgrade BillingCenter as described at “Upgrading BillingCenter from 8.0.x for ContactManager” on page 87.

This topic includes:

- “Manually Configuring Changed Files” on page 90
- “BillingCenter Web Services Version Change” on page 90

Manually Configuring Changed Files

The following web services, APIs, and helper classes changed between ContactManager 8.0.0 and 8.0.1. If you have customized any of these files or classes, you must manually merge your changes into the file at the new location.

801 Version Change to Web Service Classes, Support Classes, and WSDL Files

The ab800 node in the original packages was changed to ab801, as shown in the following list:

- gsrc/gw/webservice/ab/ab801/MaintenanceToolsAPI.gs
- gsrc/gw/webservice/ab/ab801/MessagingToolsAPI.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPI.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPIAddressSearch.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPIFindDuplicatesResult.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPIFindDuplicatesResultContainer.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPIPendingContactChange.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPIProximitySearchParameters.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPIRelatedContact.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPISearchCriteria.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPISearchResult.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPISearchResultContainer.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPISearchSortColumn.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPISearchSpec.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPISpecialistService.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPISubtypeFilter.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPITagMatcher.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPIUtil.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ABContactAPIValidateCreateContactResult.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/AddressBookUIDContainer.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/AddressBookUIDTuple.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/AddressInfo.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/ExceptionHandler.gs
- gsrc/gw/webservice/ab/ab801/abcontactapi/RelatedContactInfoContainer.gs
- gsrc/wsi/local/gw/webservice/ab/ab801/MaintenanceToolsAPI.wsdl
- gsrc/wsi/local/gw/webservice/ab/ab801/MessagingToolsAPI.wsdl
- gsrc/wsi/local/gw/webservice/ab/ab801/abcontactapi/ABContactAPI.wsdl

Changes that Support Externally Specified Unique IDs for New Contacts

The ContactManager class `gw.contactmapper.ab800.ContactMapper` has new fields that support handling external unique IDs sent from core applications. If you have customized `ContactMapper` in your configuration, you must merge your changes manually into this class. See “Mapping Fields of a ContactManager Contact” on page 244 in the *Contact Management Guide*.

BillingCenter Web Services Version Change

BillingCenter has changed the version of all its web services to bc801. Therefore, the ContactManager web services collection `bc800.wsc`, which is in `wsi.remote.gw.webservice.bc`, now uses the following path to access the WSDL file for BillingCenter:

`${bc}/ws/gw/webservice/bc/bc801/contact/ContactAPI?wsdl`

See:

- “Step 1: Integrate ContactManager with BillingCenter” on page 61 in the *Contact Management Guide*

- “Configuring ContactManager-to-BillingCenter Authentication” on page 79 in the *Contact Management Guide*

Upgrading from 7.0.x

This part describes how to perform an upgrade from BillingCenter 7.0.x to 8.0.4.

If you are upgrading from BillingCenter 8.0.x, see “Upgrading from 8.0.x” on page 27 instead.

If you are upgrading from BillingCenter 3.0.x, see “Upgrading from 3.0.x” on page 201 instead.

This part includes the following topics:

- “Upgrading the BillingCenter 7.0.x Configuration” on page 95
- “Upgrading the BillingCenter 7.0.x Database” on page 133
- “Upgrading BillingCenter from 7.0.x for ContactManager” on page 189
- “Upgrading ContactManager from 7.0.x” on page 193

Upgrading the BillingCenter 7.0.x Configuration

This topic describes how to upgrade the BillingCenter configuration from version 7.0.x to 8.0.4.

If you are upgrading from an 8.0.x version, see “Upgrading the BillingCenter 8.0.x Configuration” on page 29 instead.

If you are upgrading from a 3.0.x version, see “Upgrading the BillingCenter 3.0.x Configuration” on page 203 instead.

This topic includes:

- “Overview of ContactManager Upgrade” on page 96
- “Obtaining Configurations and Tools” on page 96
- “Creating a Configuration Backup” on page 100
- “Removing Patches” on page 100
- “Removing Language Packs” on page 100
- “Updating Infrastructure” on page 100
- “Launching the BillingCenter 8.0.4 Configuration Upgrade Tool” on page 101
- “Configuration Upgrade Tool Automated Steps” on page 102
- “Using the BillingCenter 8.0.4 Upgrade Tool Interface” on page 109
- “Configuration Merging Guidelines” on page 115
- “Data Model Merging Guidelines” on page 116
- “Changes to the Logging API” on page 121
- “Adding DDL Configuration Options to database-config.xml” on page 124
- “Merging Changes to Field Validators” on page 124
- “Renaming PCF files According to Their Modes” on page 125
- “Updating Rounding Mode Parameter” on page 125

- “Merging Display Properties” on page 125
- “Merging Other Files” on page 126
- “Fixing Gosu Issues” on page 126
- “Upgrading Rules to BillingCenter 8.0.4” on page 128
- “Translating New Display Properties and Typecodes” on page 129
- “Validating the BillingCenter 8.0.4 Configuration” on page 130
- “Building and Deploying BillingCenter 8.0.4” on page 131

Overview of ContactManager Upgrade

The automatic upgrade process for ContactManager is almost precisely the same as for BillingCenter. However, there are differences, especially for manual upgrade. Additionally, Guidewire recommends that you complete the ContactManager upgrade before manually updating files that BillingCenter uses for integration with ContactManager.

- For information on upgrading ContactManager, see “Upgrading ContactManager from 7.0.x” on page 193.
- For information on upgrading BillingCenter files used to integrate with ContactManager, see “Upgrading BillingCenter from 7.0.x for ContactManager” on page 189.

Obtaining Configurations and Tools

Configuration refers to everything related to the application except the database. This includes configuration files such as typelists and PCF files, the file structure, web resources, Gosu classes, rules, plugins, libraries, localization files, and application server files.

The upgrade process involves three configurations. This guide defines and refers to these configurations as base, customer, and target.

Base – The unedited, original configuration on which you based your customer configuration. The base configuration is included in directories within `/modules` other than `/configuration`.

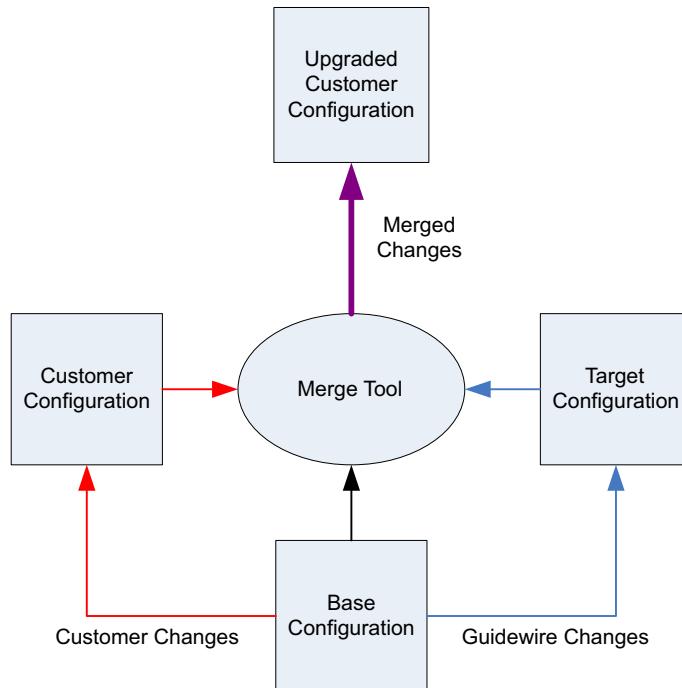
Customer – The configuration you are now using and will upgrade. This is the base configuration of the BillingCenter version that you currently run with your custom configuration applied. Custom configuration files are stored in the `modules/configuration` directory. The Configuration Upgrade Tool compares your customized versions of files in the `modules/configuration` directory and its subdirectories with files in the target BillingCenter version. If you have customized files outside of `modules/configuration` and its subdirectories, manually upgrade those files by comparing them with versions of those files in the target BillingCenter version.

Target – The unedited, original configuration of BillingCenter 8.0.4 on which your upgraded configuration will be based. Guidewire grants you access to the Guidewire Resource Portal, from which you download the target configuration ZIP file. Unzip the target BillingCenter 8.0.4 configuration into another directory. Download the latest patch release for the target version that you are downloading. Follow the instructions with the patch release to install it after you unzip the target version. Do not make any modifications to the target configuration prior to completing the configuration upgrade. Do not start Guidewire Studio for the target configuration until you have completed the configuration upgrade.

IMPORTANT Set all files in the base, customer, and target configurations to writable before beginning the upgrade.

The following figure shows how you use these configurations to create a merged configuration. The merged configuration combines your changes to the original base configuration (the customer configuration) and Guidewire

changes to the base configuration (the target configuration). The original base configuration provides a basis for comparison.



Viewing Differences Between Base and Target Releases

To view an inventory of the differences between the base release and the target release, download and carefully review the *Upgrade Diffs Report* from the Guidewire Resource Portal.

1. Open a browser to <https://guidewire.hivelive.com/pages/home>.
2. Click Project Center → Upgrade Services.
3. Click Review the Upgrade Diff Reports.
4. Click BillingCenter.
5. Click Upgrade Diff Reports - BillingCenter or Upgrade Diff Reports - ContactManager.
6. Click Upgrade From *base version*.
7. Click Upgrade To 8.0.4.

Specifying Configuration and Tool Locations

The BillingCenter 8.0.4 Configuration Upgrade Tool depends on the following tools:

- **Text Editor** – An ASCII text editor you use to edit programs and similar files, for example, Notepad, WordPad or Textpad. This editor must not put additional characters in files, as Word does.

- **Merge Tool** – An editor which displays two or three versions of a file, highlights the differences between them, and allows you to perform basic editing functions on them. Also known as a “diff tool.” Examples include Araxis Merge Professional and Beyond Compare Professional. If using Beyond Compare Professional, see “Considerations for Using Beyond Compare Professional” on page 99. Configure the merge tool to ignore end of line characters to reduce the number of potential false positives during the configuration upgrade step.

IMPORTANT The merge tool that you use must support three-way file comparison and merging. During the configuration upgrade, for some files you will need to compare three versions: the original base version, the new version and your customized version.

The Configuration Upgrade Tool needs the location of the BillingCenter environment that you will upgrade. The tool stores all versions of files to be merged and merge results in a `tmp` directory that it creates within the target environment. Define paths to the configuration and tools in the `BillingCenter/modules/ant/upgrade.properties` file of the target BillingCenter 8.0.4 environment. Remove the pound sign, ‘#’, preceding each property to uncomment the line and then specify values. Use double backslashes in paths. For example, `C:\\\\BillingCenter`. You do not need to use quotes for paths that include spaces.

The following properties are configurable in `upgrade.properties`.

Property	Description
<code>upgrader.priorversion.dir</code>	Path to the top-level BillingCenter directory of the current customer environment. This directory contains <code>/bin</code> and other BillingCenter directories.
<code>upgrader.editor.tool</code>	Path to an executable editing tool.
<code>upgrader.diff.tool</code>	Path to an executable difference editor tool, such as Araxis Merge Professional or Beyond Compare Professional, also known as a merge tool, used for two-way merges. If your difference editor supports both two and three-way merges, you can use the same value for <code>upgrader.diff.tool</code> and <code>upgrader.merge.tool</code> .
<code>upgrader.merge.tool</code>	Path to an executable difference editor tool, such as Araxis Merge Professional or Beyond Compare Professional, also known as a merge tool, used for three-way merges. The merge tool specified for <code>upgrader.merge.tool</code> must support three-way file comparison and merging. If your difference editor supports both two and three-way merges, you can use the same value for <code>upgrader.diff.tool</code> and <code>upgrader.merge.tool</code> . You might need to configure the display of your merge tool to show three panels.

Property	Description
upgrader.merge.tool.arg.order	<p>The order of command line arguments to the difference editor tool specified by upgrader.merge.tool.</p>
	<p>The command line arguments available to the difference editor typically include the display order, from left to right, for versions of a file viewed in the difference editor.</p>
	<p>The available options are:</p>
	<p>NewBase is the unmodified target version provided with BillingCenter 8.0.4.</p>
	<p>PriorBase is the original base version.</p>
	<p>PriorCustom is your configured version.</p>
	<p>Resulting is the merged output file.</p>
	<p>The order of these values controls the display order in the difference editor tool. If the tool displays just two versions, it uses the same relative order.</p>
	<p>By default, the display order places the old base version of a file in the center. The old base version is the common ground between the new uncustomized version and the old customized version. Guidewire changed the old base version to create the new target version, and you changed the old base version to create the customized version in your configuration. With the old base version in the center, you can incorporate both sets of changes to create a customized target version.</p>
	<p>The default order enables you to merge changes from the left and right to the center and save the merged version. If you use another difference editor, you might need a different order to achieve the same result. Samples are shown below for various difference engines:</p>
	<p>Araxis Merge Professional</p>
	<pre>upgrader.merge.tool.arg.order = NewBase PriorBase PriorCustom</pre>
	<p>Beyond Compare Professional</p>
	<pre>upgrader.merge.tool.arg.order = NewBase PriorCustom PriorBase Resulting</pre>
	<p>P4Merge</p>
	<pre>upgrader.merge.tool.arg.order = PriorBase NewBase PriorCustom</pre>
	<p>You might need to configure the display of your merge tool to show three panels.</p>
upgrader.steps.class	<p>The class to run to execute the configuration upgrade automated steps. If you are upgrading BillingCenter 3.0 or newer, then leave this property commented out.</p>
exclude.pattern	<p>A regular expression pattern for paths of files for the Configuration Upgrade Tool to mark as unmergeable. Typically, you use exclude.pattern to specify source control metadata files. Samples are provided in upgrade.properties for CVS and SVN.</p>

Considerations for Using Beyond Compare Professional

By including the Resulting parameter on the upgrader.merge.tool.arg.order property for Beyond Compare, you can avoid an extra manual step to specify the output file on each merge.

To configure the Configuration Upgrade Tool for Beyond Compare Professional

1. Create a new file in a text editor.
2. Add the following text to the new file, modifying the location of the Beyond Compare Professional executable if necessary.


```
@echo off
"C:\Program Files (x86)\Beyond Compare 3\BCompare.exe" %1 %2 %3 %4
```
3. Save the file as merge.cmd in the BillingCenter/modules/ant directory.
4. Open the BillingCenter/modules/ant/upgrade.properties file of the target BillingCenter 8.0.4 environment.

5. Set the `upgrader.diff.tool` property to the location of the Beyond Compare Professional executable. For example:
`upgrader.diff.tool = "C:\\Program Files (x86)\\Beyond Compare 3\\BCompare.exe"`
6. Set the `upgrader.merge.tool` property to a command line script file. For example:
`upgrader.merge.tool = ".\\modules\\ant\\merge.cmd"`
7. Set the `upgrader.merge.tool.arg.order` property:
`upgrader.merge.tool.arg.order = NewBase PriorCustom PriorBase Resulting`
8. Set the remaining properties in `upgrade.properties` as described in the table.
9. Save `upgrade.properties`.

Creating a Configuration Backup

Prepare the environment so that you can make a total recovery of the original installation if you run into problems during the upgrade.

Guidewire recommends that you track BillingCenter configuration changes in a source code control system. Before upgrading, have a labeled version of your entire pre-upgrade BillingCenter configuration folder (`modules/configuration`) from your custom configuration. A labeled version is a named collection of file revisions.

As an even stronger precaution, make a backup of the same installation directories.

Removing Patches

If you have applied any patches from Guidewire to BillingCenter, remove the patches before you run the configuration upgrade. Patches are specific to the pre-upgrade version. If you do not remove the JAR files for patches, the Configuration Upgrade Tool copies the JAR files to the upgraded configuration. In that case, your upgraded configuration will be using a JAR file that is not current. Patches are typically installed by adding JAR files within `modules/configuration/deploy`.

Removing Language Packs

If you have language packs installed, you must remove the language packs before upgrading BillingCenter. See “Upgrading Display Languages” on page 30 in the *Globalization Guide*.

Updating Infrastructure

Before starting the upgrade, have the supported server operating systems, application server and database software, JDK, and client operating systems for the target version. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

For SQL Server, after you upgrade the database server software, run the following command to set the compatibility level:

```
ALTER DATABASE databaseName SET COMPATIBILITY_LEVEL = 110
```

Launching the BillingCenter 8.0.4 Configuration Upgrade Tool

To upgrade your configuration, merge Guidewire changes to the base configuration with your changes. The *Configuration Upgrade Tool*, provided by Guidewire with the target BillingCenter 8.0 configuration, facilitates this process. The tool compares your customized versions of files in the `modules/configuration` directory and its subdirectories with files in the target BillingCenter version. If you have customized files outside of `modules/configuration` and its subdirectories, manually upgrade those files by comparing them with versions of those files in the target BillingCenter version.

The Configuration Upgrade Tool requires two tools: a merge tool such as Araxis Merge Professional or Beyond Compare 3 Professional, and a text editor. Configure the merge tool to ignore end-of-line characters to reduce the number of potential false positives during the configuration upgrade step.

IMPORTANT The merge tool that you use must support three-way file comparison and merging. During the configuration upgrade, for some files you will need to compare three versions: the original base version, the new version and your customized version.

The Configuration Upgrade Tool performs a series of automated steps and then opens an interface that you use for the manual merge process.

Guidewire can provide guidance on using the Configuration Upgrade Tool in a multi-user environment using a source control management system.

See the *Upgrade Diffs Report* for an inventory of the differences between the base release and the target release. To retrieve the *Upgrade Diffs Report* follow the procedure described in “Viewing Differences Between Base and Target Releases” on page 97.

Also see the *BillingCenter New and Changed Guide* for a description of new features and changes to existing features. Review key data model changes as these changes might impact customizations in your system.

To launch the Configuration Upgrade Tool

1. Open a command window.
2. Navigate to the `modules/ant` directory of the target configuration.
3. Execute the following command:

```
ant -f upgrade.xml upgrade > upgrade_log.txt
```

You can specify any file to log messages and exceptions.

The Configuration Upgrade Tool first copies the modules of the base environment to a `tmp/cfg-upgrade/modules` directory in the target environment. The base environment is specified by the `upgrader.priorversion.dir` property in `modules/ant/upgrade.properties` in the target environment.

The Configuration Upgrade Tool then performs a number of automated steps, described later in this topic. Once the tool completes the automated steps, it opens a user interface. The interface opens whether the automated steps were successful or not. Review the log file or console before proceeding with the manual merge process.

Restarting the Configuration Upgrade Tool

The Configuration Upgrade Tool stores work in progress by recording which files you have marked resolved in the `accepted_files.1st` file. This file is stored in the `merge` directory of the target environment. You can close the interface and restart it later without losing your work in progress.

If you do want to start the upgrade over, use the `clean` command to empty the working directories.

```
ant -f upgrade.xml clean
```

WARNING If you empty the `tmp` directory after beginning to merge, you lose all completed merges that you have not resolved and moved into the target configuration directory.

Configuration Upgrade Tool Automated Steps

The Configuration Upgrade Tool prepares for the manual configuration merge process by performing a number of automated steps. Review these steps before proceeding with the configuration merge. Understanding these automated steps helps to understand some file changes you will see when merging the configuration. Finally, some steps might require manual intervention if there is an issue.

Removing Template Pages

The Configuration Upgrade Tool deletes PCF template pages. These pages have a `<TemplatePage>` root element. The upgrade also removes `<EntryPoint>` elements that reference template pages. Template pages have been replaced by SOAP-based data integration in BillingCenter 8.0. See “Template Page PCF Files Removed” on page 42 in the *New and Changed Guide*.

Updating PCF Files

The Configuration Upgrade Tool performs the following modifications to PCF files:

- Removes the `reflectOnBottom` attribute. This attribute was used to display the a virtual toolbar at the bottom of a page. The attribute was removed because the user interface needs to match the server configuration. No alternative configuration is available.
- Converts all `postOnChange` attributes on a value widget to a child `PostOnChange` node. For example, the upgrade converts:

```
<Input id="xxx" postOnChange="true" onChange="someMethod()" disablePostOnEnter="doEvaluation()"/>
```

to:

```
<Input id="xxx">
  <PostOnChange onChange="someMethod()" disablePostOnEnter="doEvaluation()"/>
</Input>
```
- Removes the `showNoneSelected` attribute from all `DetailView` inputs that are bound to a value. Setting `showNoneSelected=false` would suppress the **None Selected** option from drop-down lists and would default to the first option. This type of configuration was incorrect because the selection of the option was generally programmatically incorrect and was often used as a shortcut instead of specifying an explicit default. Verify all removals to ensure there is not any dependent logic. If there is, specify an explicit default in the page configuration.
- Removes the `showNoneSelected` attribute from all `<ValueCellType>` nodes. See the above note about removal of the `showNoneSelected` attribute from all `DetailView` inputs that are bound to a value.
- Removes the `numDataEntriesPerRow` and `transposed` attributes from `RowIteratorNode` elements. Transposed lists are a relatively rare configuration. If you had one in your configuration, use a traditional list view.
- Removes `<DetailViewPanel>` elements from `<ButtonCell>`, `<ButtonInput>`, and `<ToolbarButtonType>` elements. Detail views can no longer be embedded inside buttons.
- Converts `valueWidth` attributes on cell widgets to `value` attributes. As of 8.0, BillingCenter sizes cells by heuristics rather than content, so `valueWidth` is no longer necessary.
- If all cells in a row have the `useHeaderStyle="true"` property, the upgrade moves the property to the row level. A list can only have one header. See below.

- Updates rows to rename the `useHeaderStyle` property to `renderAsSmartHeader`. The property is renamed because the header functionality is more than styling. When a row is rendered as a smart header, all the row header interactive features are made available.
- Renames `<ContentCell>` elements to `<Link>`.
- Converts `<Cell>` elements within `<ColumnFooter>` to `<TextCell>` elements.
- Removes any element that is not a `<TextCell>` element from `<ColumnFooter>` elements.
- Removes `<ColumnHeader>` elements from `<CellType>` elements.
- Remove `<DetailViewPanel>` from `<ContentCell>`. The upgrade performs the following steps. After the automatic upgrade, review your `<ContentCell>` configurations to manually verify the configuration and make any changes. Content cells cannot have editable detail views embedded in them. Review all removals to ensure functionality. If editable content is needed within a row of data, the recommended configuration is a list detail panel.
 - For any `<ContentCell>` that contains a `<DetailViewPanel>`, the upgrade renames the `<ContentCell>` to `<FormatCell>`.
 - For other types of `<ContentCell>`, the upgrade renames the element to `<LinkCell>`.
 - Removes elements that are not allowed in the `<FormatCell>`, such as `<DetailViewPanel>` and `<InputColumn>`. This strips out unnecessary container elements. No content will be removed.
 - Renames inputs in the `<DetailViewPanel>` to `<TextInput>` unless they are `<ContentInput>`, `<TextInput>`, or `<NoteBodyInput>`.
 - Removes attributes that were allowed on specific input elements but not on `<TextInput>`.
- Removes the `useHeaderStyle` attribute from all cells that can be bound to a value. The header style in 8.0 is a lot more extensive. Smart header capabilities have been added, in addition to the styling. Header capabilities are at the row level as opposed to the cell. If you are interested in highlighting content, there are a few other ways to achieve that. Review the PCF reference for a full list of attributes for that particular cell variant.
- Removes the `compress` attribute from `<DetailViewPanel>`.
- Removes the `compress` attribute from `<ListViewPanel>`.
- Removes the `compressIfSingleChild` attribute from `<InputGroup>`.
- Comments out `<ProgressCell>` elements. This was an uncommon widget that Guidewire has removed. If you were using it on some page and would like to continue to do so, create a list detail panel, and use the `ProgressInput` in the detail section instead.
- Removes the `refreshOnProgressComplete` attribute from `<ListViewPanel>` and `<Row>` elements. This is part of the removal of the `<ProgressCell>` widget.
- Removes the following attributes from `<ChartPanel>`:
 - `bgColor`
 - `border`
 - `displayPlotOutline`
 - `orientation`
 - `sameSeriesColor`
 - `threeD`
 - `tooltip`
- Guidewire cleaned up the `<ChartPanel>` schema as a part of simplification and a move to a more interactive experience.
- Removes the following attributes from `<DomainAxis>`:
 - `autoRange`
 - `autoRangeIncludesZero`
 - `tickUnit`

- `upperMargin`
- Removes the `<Interval>` element.
- Removes the following attributes from `<RangeAxis>`:
 - `autoRange`
 - `autoRangeIncludesZero`
 - `tickUnit`
 - `upperMargin`
- Removes the `percentComplete` attribute from `<DataSeries>`.
- Removes the following from `<DualAxisDataSeries>`:
 - `autoRangeIncludesZero`
 - `lowerMargin`
 - `tickUnit`
 - `tooltip`
 - `upperMargin`
- Removes the following chart types from the `<ChartType>` enumerator:
 - `Waterfall`
 - `Gantt`
- Renames the following chart types in the `<ChartType>` enumerator:
 - `Dial` → `Gauge`
 - `Polar` → `Radar`
 - `Ring` → `Pie`
 - `StackedArea` → `Area` (there is no more distinction between a stacked vs non-stacked area)
 - `XYStep` → `XYLine`
 - `XYStepArea` → `XYArea`

Upgrading Work Queue Configuration

The Configuration Upgrade Tool makes the following changes to `work-queue.xml`:

- removes obsolete `minpollinterval` attribute.
- removes obsolete `orphansFirst` attribute.
- removes obsolete `checkInAfterError` attribute
- adds `retryInterval=va7ue` (the upgrade sets the value to 0 if `checkInAfterError` was `true`, or to the current value of `progressinterval` if `checkInAfterError` was not `true`.)

For more information about changes to work queue configuration, see “Changes to Work Queue Configuration” on page 76 in the *New and Changed Guide*.

Upgrading Database Configuration

The Configuration Upgrade Tool moves the database configuration from `config.xml` to `database-config.xml` and converts it to the BillingCenter 8.0 format.

As of BillingCenter 8.0, Guidewire has made the following changes to the database configuration:

- The `<database>` element no longer contains subelements with the following syntax:
`<param name="name" value="va7ue">`

- For Oracle, the `<tablespacemapping>` elements have been replaced with a single `<tablespaces>` element. The `<tablespaces>` element is contained in an `<ora-db-ddl>` parent element. The `<tablespaces>` element includes the attributes `admin`, `index`, `op`, `staging`, `typelist`, and `lob`. These attributes correspond to the logical tablespaces defined in BillingCenter. You can use these attributes to map tablespaces that you have created to the logical tablespaces.
- For SQL Server, the `<tablespacemapping>` elements have been replaced with a single `<mssql-filegroups>` element. The `<mssql-filegroups>` element is contained in an `<mssql-db-ddl>` parent element. The `<mssql-filegroups>` element includes the attributes `admin`, `index`, `op`, `staging`, `typelist`, and `lob`. These attributes correspond to the logical tablespaces defined in BillingCenter. You can use these attributes to map file groups that you have created to the logical tablespaces.
- If a `<tablegroup>` element was contained in a `<database>` element that had an `env` attribute defined, the upgrade copies the `env` attribute onto the `<tablegroup>` element.
- If any of the following `<database>` attributes are defined, the upgrade copies them over to the `<database>` element in `database-config.xml`: `addforeignkeys`, `autoupgrade`, `checker`, `dbtype`, `env`, `name`, `printcommands`. The schema for these attributes has not changed.
- If any comments exist within the `<database>` element, the upgrade copies these comments to the `<database>` element in `database-config.xml`.
- If the `driver` attribute of the `<database>` element equals `dbcp`, the upgrade adds a `<dbcp-connection-pool>` element and copies the `jdbcUrl` parameter to the `jdbc-url` attribute of the `<dbcp-connection-pool>` element. If the original configuration did not include a `jdbcUrl` parameter, then the upgrade logs an error. If a `passwordFile` attribute is specified on the `<database>` element of the old configuration, the upgrade transfers the `passwordFile` attribute to the `<dbcp-connection-pool>` element. The upgrade converts any of the following parameters defined in the old configuration to attributes on the `<dbcp-connection-pool>` element:
 - `maxActive`
 - `maxIdle`
 - `maxWait`
 - `minEvictableIdleTimeMillis`
 - `numTestsPerEvictionRun`
 - `testOnBorrow`
 - `testOnReturn`
 - `testWhileIdle`
 - `timeBetweenEvictionRunsMillis`
 - `whenExhaustedAction`
- If the `driver` attribute of the `<database>` element equals `dbcp` and any of the following attributes are set, the upgrade creates a `<reset-tool-params>` element within the `<dbcp-connection-pool>` element:
 - `collation`
 - `oracle.tnsnames`
 - `system.username`
 - `system.password`

The upgrade then transfers any of these attributes that are defined to the new `<reset-tool-params>` element.

- If the `driver` attribute of the `<database>` element equals `jndi`, the upgrade adds a `<jndi-connection-pool>` element and copies the `jndi.datasource.name` parameter to the `datasource-name` attribute of the `<jndi-connection-pool>` element. If the original configuration did not include a `jndi.datasource.name` parameter, then the upgrade logs an error.
- If the old configuration includes an `<upgrade>` element within the `<database>` element, the upgrade adds an `<upgrade>` element to the `<database>` element of the new configuration.
- If the old configuration contains an `<upgrade>` element that includes an `oracleMarkColumnsUnused` attribute, the upgrade converts the attribute to a `deferDropColumns` attribute, preserving the value.

- If the old configuration contains an `<upgrade>` element that includes a `verifySchema` attribute, the upgrade copies this attribute to `<upgrade>` element of the new configuration.
- If the old configuration contains an `<upgrade>` element that contains an `<oracleddloptions>` or `<sqlserverddlopts>` element, the upgrade logs a warning. You must upgrade these elements manually.
- If the old configuration includes a `<databasestatistics>` element within the `<database>` element, the upgrade copies the `<databasestatistics>` element to the `<database>` element of the new configuration.
- For Oracle databases, if the `<database>` element includes any of the following parameters, the upgrade creates an `<oracle-settings>` element within the `<database>` element of the new configuration:
 - `queryRewriteEnabled`
 - `statisticsLevel`
 - `stored.outlines`
 - `UseDbResourceMgrCancelSql`

The upgrade converts any of the above parameters to attributes on the new `<oracle-settings>` element. The attributes have the following names:

- `query-rewrite`
- `statistics-level-all` (if any value is set for `statisticsLevel` in the old configuration, the upgrade sets the `statistics-level-all` attribute to `true` in the new configuration. The value `ALL` was the only valid value for the `statisticsLevel` parameter in the old configuration.)
- `stored-outline-category`
- `db-resource-mgr-cancel-sql`
- For SQL Server databases, if the `<database>` element includes either the `msjdbctracelevel` or `msjdbctracefile` parameter, the upgrade adds a `<sqlserver-settings>` element within the `<database>` element of the new configuration. The upgrade then converts the `msjdbctracelevel` and `msjdbctracefile` parameters to `jdbc-trace-level` and `jdbc-trace-file` attributes on the `<sqlserver-settings>` element.
- For SQL Server databases, if the `unicodecolumns` parameter is defined in the old configuration, the upgrade adds a `unicodecolumns` attribute to the `<sqlserver-settings>` element of the new configuration. If the `<sqlserver-settings>` element has not yet been created, the upgrade creates the element.
- If any `<tablespacemapping>` elements are defined in the old configuration, the upgrade creates an `<upgrade>` element within the `<database>` element of the new configuration if one does not yet exist. The upgrade then does the following, depending on the database type:
 - For Oracle, the upgrade adds an `<ora-db-ddl>` element within the `<upgrade>` element of the new configuration, if an `<ora-db-ddl>` element is not yet defined. The upgrade then adds a `<tablespaces>` element to the `<ora-db-ddl>` element and converts each `<tablespacemapping>` element to an attribute on the `<tablespaces>` element. The upgrade then adds an `<ora-lob>` element to the `<ora-db-ddl>` element and sets the `<ora-lob>` attribute type to `BASICFILE`. Although Oracle 12c creates SecureFile LOB columns by default, the configuration upgrade sets the default type for any new LOBs to `BASICFILE` to maintain consistency with the Oracle 11 default. If Oracle LOBs are configured to be SecureFile or compressed SecureFiles, the configuration upgrade does not transfer the DDL settings to `database-config.xml`. These configuration settings must be applied to the new `database-config.xml` database element manually. Note that if you change a DDL configuration, the setting only applies for new objects.
 - For SQL Server, the upgrade adds an `<mssql-db-ddl>` element within the `<upgrade>` element of the new configuration, if an `<mssql-db-ddl>` element is not yet defined. The upgrade then adds a `<mssql-filegroups>` element to the `<mssql-db-ddl>` element and converts each `<tablespacemapping>` element to an attribute on the `<mssql-filegroups>` element.

Splitting Localization.xml into Separate Files for each Locale

The upgrade splits the locales from the single `localization.xml` file used in BillingCenter 7.0 into a separate file for each locale defined by a `<GWLocale>` element. The new location for each split `localization.xml` file is `config/locale/locale/`. Each `localization.xml` file can have only one `GWLocale` element in BillingCenter 8.0.

Splitting address-config.xml into Separate Files for each Country

The upgrade splits the address format definitions from the single `address-config.xml` file used in BillingCenter 7.0 into a separate file for each country defined by an `<AddressDef>` element. The new location for each split `address-config.xml` file is `config/geodata/country code/`. Each `address-config.xml` file can have only one `AddressDef` element in BillingCenter 8.0.

Splitting zone-config.xml into Separate Files for each Country

The upgrade splits the zone configuration definitions from the single `zone-config.xml` file used in BillingCenter 7.0 into a separate file for each country. Zones for each country are defined by a `<Zones>` element with a `countryCode` attribute. The new location for each split `zone-config.xml` file is `config/geodata/country code/`. In BillingCenter 8.0 each `zone-config.xml` file can have only one `<Zones>` element that contains zones for a single country.

Splitting currencies.xml into Separate Files for each Currency

The upgrade splits the currency definitions from the single `currencies.xml` file used in BillingCenter 7.0 into a separate file for each currency type. Each currency type is defined by a `<CurrencyType>` element with a `code` attribute. The separate files are each named `currency.xml`. The new location for each `currency.xml` file is `config/currencies/code/`, where `code` is the value of the `code` attribute on the `<CurrencyType>` element.

Moving Country-based Field Validator Definition Files

The upgrade moves each country-based field validator definition file to an individual directory. Country-specific field validator definition files are named with the format `fieldvalidators_country code.xml`, such as `fieldvalidators_JP.xml` for field validators specific to Japan. The upgrade moves each country-specific field validator definition file to `config/fieldvalidators/country code/`. The generic `fieldvalidators.xml` file remains at `config/fieldvalidators/`.

Moving Rules Files up One Directory

The upgrade moves all rules files up one directory from `config/rules/rules/` to `config/rules/`.

Reformatting Rules for Display in Studio Rules Editor

The upgrade reformats `.gr` rule files so that the Studio rules editor recognizes the file contents as rules.

Copying Custom Rules and Adding BillingCenter 8.0.4 Default Rules

The upgrade copies customized rules to the target configuration `modules/configuration/config/rules` directory.

This step also copies the default rules provided with BillingCenter 8.0.4 to a BillingCenter 8.0.4 folder within the `modules/configuration/config/rules` directory of the target configuration. This is so you have a copy of the default rules in a folder in Studio that you can use to compare with your custom rules.

Renaming SOAP Web Services from XML to RWS

The upgrade changes the extension of SOAP web service files in config/webservices from .xml to .rws.

Renaming Plugins from XML to GWP

The upgrade changes the extension of plugin files in config/plugin/registry from .xml to .gwp.

Renaming Display Names Files from XML to EN

The upgrade changes the extension of display names files in config/displaynames from .xml to .en.

Upgrading Display Keys

The upgrade compares display keys from the custom configuration with display keys in the base 7.0 configuration and display keys in the default BillingCenter 8.0 configuration. The following display key files are inspected.

- display.properties
- gosu.display.properties
- productmodel.display.properties
- studio.display.properties
- typelist.properties

The upgrade compares the case of display property keys in the custom configuration with the case of the key in the default BillingCenter 8.0.4 configuration. If the case does not match, but the value assigned to the key matches the value in the default configuration, the upgrade corrects the case in the custom configuration. If the case of the keys does not match, and the value is different in the custom configuration, the upgrade reports an error.

The upgrade then merges the display keys files into a single file for each locale. This file has the extension .merged. The merged display properties files are available in the Configuration Upgrade Tool for comparison with the default BillingCenter display.properties. You can merge Guidewire changes and new properties with your custom properties values.

Adding nullok="true" to Entity and Extension Foreign Key Columns

The upgrade modifies ETI and EIX files in config/metadata and ETX and ETI files in config/extensions. The upgrade adds the attribute nullok="true" to <foreignkey> and <edgeForeignKey> elements if the element did not explicitly specify a value for the nullok attribute. In BillingCenter 8.0, the nullok attribute is required to be explicitly set.

Removing deletefk Attribute from Entity and Extension Foreign Keys

The upgrade removes the deletefk attribute from all <foreignkey> and <edgeforeignkey> elements that include a deletefk attribute.

Setting XML Namespace on Metadata Files

This step sets the XML namespace on data model and typelist entity and extension files in config/metadata and config/extensions to <http://guidewire.com/datamodel> and <http://guidewire.com/typelists> respectively. You can configure an XML editor to map these namespaces to XSD files that define the structure of data model and typelist files. Map <http://guidewire.com/datamodel> to BillingCenter/modules/p1/xsd/metadata/datamodel.xsd and <http://guidewire.com/typelists> to BillingCenter/modules/p1/xsd/metadata/typelists.xsd. Then, the XML editor can validate entities as you create or modify them.

The namespace was encouraged but optional prior to 8.0. The namespace must be specified in 8.0.

Upgrading Document Assistant Parameters

In BillingCenter 8.0, Guidewire Document Assistant uses a Java applet deployed using JNLP instead of an ActiveX control. The upgrade updates `config.xml` for this change. In this step, the upgrade replaces legacy Document Assistant ActiveX configuration parameters with the updated ones. The upgrade makes the following changes:

- Renames `AllowActiveX` to `AllowDocumentAssistant`, ignoring the old value. In 8.0 `AllowDocumentAssistant` defaults to `false`, whereas `AllowActiveX` was `true` in prior releases. The deployment, security, and configuration of applets is entirely different from ActiveX controls. Consider Java security issues as part of your decision to deploy the Document Assistant applet.
- Renames `UseGuidewireActiveXControlToDisplayDocuments` to `UseDocumentAssistantToDisplayDocuments`, keeping the old value.
- Removes `AllowActiveXAutoInstall`.
- Removes `UseDocumentNameAsFileName`.
- Adds `DocumentAssistantJNLP`.

See “Document Creation and Document Management Parameters” on page 41 in the *Configuration Guide*.

Separating Entities and Typelists

The upgrade creates `entity` and `typelist` folders in `config/metadata` and `config/extensions` directories. The upgrade then moves ETI, EIX, and ETX files into the `entity` folders and moves TTI, TIX, and TTX files into the `typelist` folders.

Using the BillingCenter 8.0.4 Upgrade Tool Interface

IMPORTANT Review the automated step descriptions before you proceed. Some automated steps might require you to perform a manual step while merging the configuration. Typically, such automated steps insert a warning into a file. Check the `steps_results.txt` file for warning and error messages. Correct any issues reported. Then, delete `steps_results.txt` and restart the Configuration Upgrade Tool.

After the Configuration Upgrade Tool completes the automated steps, the working area contains up to three versions of the same file:

- The *customer* file.
- The *base* file, from which you configured the customer file.
- The *target* file, from BillingCenter 8.0.4.

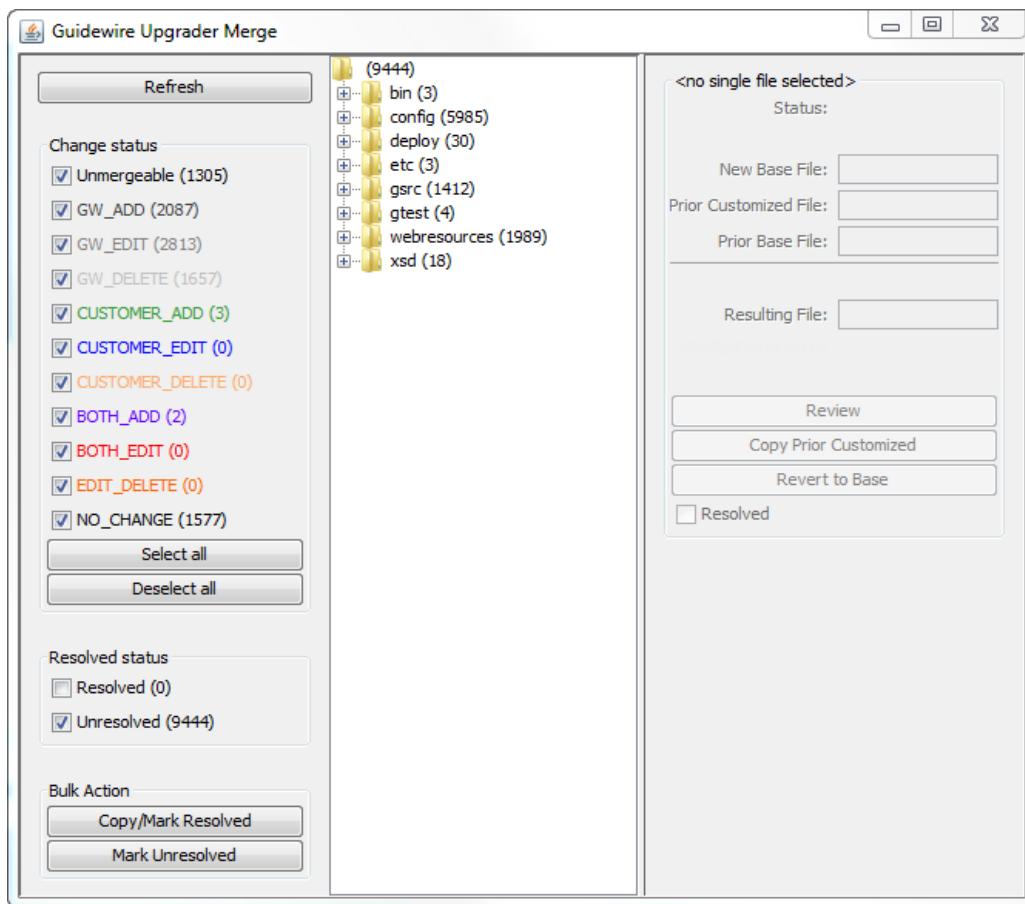
In the manual process of the upgrade, you decide whether to use one of these versions unchanged, or merge versions together. The Configuration Upgrade Tool provides a user interface to assist with the manual process. This interface has several important functions:

- It shows a complete list of all configuration files.
- It allows you to filter this list. You can, for example, view a list of all files that differ between the target version and your version. See “Change Status Filters” on page 111.
- It displays two or three versions of a file and their differences, using a merge tool you supply, such as Araxis Merge or P4Merge, defined in `upgrade.properties`.

- It lets you edit your file, incorporating changes from the other file versions, and save it.
- It lets you accept this merged version instead of one of the previous versions.
- It lets you edit the file after you have accepted changes from the merge using the text editor defined in `upgrade.properties`.

After you have accepted or merged all files that the Configuration Upgrade Tool displays, the merging process is complete.

The Configuration Upgrade Tool displays three panels. The center panel is a tree view of the files in the configuration, filtered by filter choices selected in the left panel. Files appear in the color of the filter that found them. As you select a file in the center panel, the right panel displays file information and buttons to perform actions on that file.



Filters

The left panel of the Configuration Upgrade Tool contains:

- Refresh Button
- Change Status Filters
- Resolved Status Checkboxes
- Bulk Action Buttons

Refresh Button

If multiple users are working in the same directory, each user can mark files as resolved. The Refresh button refreshes the resolved status of files shown in the Configuration Upgrade Tool for changes contributed by all users working in the same directory.

Change Status Filters

This table lists the change status filters that the Configuration Upgrade Tool displays in the left panel. Use the check boxes next to the filters to select one or any combination of change statuses to view. Use the **Select all** or **Deselect all** buttons to select or deselect all filters. The following table describes change status filters. The Guidewire Action column lists the change Guidewire has made to files matching a status filter since the prior version. The Your Action column lists the change to the file in your implementation:

Merge Status	Guidewire Action	Your Action	Type of change made to file	Action taken by Configuration Upgrade Tool
Unmergeable	change format of file	any	file exists in a different format and thus cannot be merged with an old version	If you resolve the file, the Configuration Upgrade Tool takes no action. The file, in the new format, already exists in the target configuration. The Configuration Upgrade Tool automatically marks certain files as unmergeable, including rules files. The Configuration Upgrade Tool upgrades these files before the interface displays. You can also specify a regular expression pattern in upgrade.properties for file paths to mark files matching that pattern as unmergeable. Set the pattern as the value of the exclude.pattern property.
GW_ADD	add	none	file in target not in base	If you resolve the file, the Configuration Upgrade Tool takes no action. The file added by Guidewire already exists in the target configuration. Double-clicking opens the file in the text editor specified by upgrader.editor.tool in upgrade.properties.. If you make changes, the tool prompts you to copy the file to the target configuration.
GW_EDIT	edit	none	file in target differs from base	If you resolve the file, the Configuration Upgrade Tool takes no action. The file added by Guidewire already exists in the target configuration. Double-clicking opens the file in the merge tool specified by upgrader.diff.tool in upgrade.properties to perform a comparison between the new Guidewire version and the original base version. If you make changes, the tool prompts you to copy the file to the target configuration.
GW_DELETE	delete	none	file in base not in target	If you resolve the file, the Configuration Upgrade Tool takes no action. The file deleted by Guidewire no longer exists in the target configuration. Double-clicking opens the file in the text editor specified by upgrader.editor.tool in upgrade.properties.. If you make changes, the tool prompts you to copy the file to the target configuration.
CUSTOMER_ADD	none	add	file in customer configuration only	If you resolve the file, the Configuration Upgrade Tool copies the file to the target configuration if the file has not been copied there already. Double-clicking opens the file in the text editor specified by upgrader.editor.tool in upgrade.properties.. If you make changes, the tool prompts you to copy the file to the target configuration.

Merge Status	Guidewire Action	Your Action	Type of change made to file	Action taken by Configuration Upgrade Tool
CUSTOMER_EDIT	none	edit	file differs between customer and base configurations	If you resolve the file, the Configuration Upgrade Tool copies the file to the target configuration if the file has not been copied there already.
			file unchanged between base and target configurations	Double-clicking opens the file in the merge tool specified by <code>upgrader.diff.tool</code> in <code>upgrade.properties</code> to perform a comparison between your custom version and the original base version. If you make changes, the tool prompts you to copy the file to the target configuration.
CUSTOMER_DELETE	none	delete	file exists in the base and target configurations but not in the customer configuration	If you click Delete , the Configuration Upgrade Tool removes the file from the target configuration. If you click Revert to Base , the Configuration Upgrade Tool leaves the file in the target configuration.
BOTH_ADD	add	add	new file with matching name in both target and customer configurations (rare)	You must either merge the two versions of the file or copy your prior version of the file into the target configuration before you can resolve the file. Double-clicking opens the file in the merge tool specified by <code>upgrader.diff.tool</code> in <code>upgrade.properties</code> to perform a merge between your version and the Guidewire version. If you make changes, the tool prompts you to copy the merged file to the target configuration.
BOTH_EDIT	edit	edit	file changed in both customer and target configurations	You must either merge the two versions of the file or copy your prior version of the file into the target configuration before you can resolve the file. Double-clicking opens the file in the merge tool specified by <code>upgrader.merge.tool</code> in <code>upgrade.properties</code> to perform a three-way merge between your custom version and the updated Guidewire version. If you make changes, the tool prompts you to copy the merged file to the target configuration.

Merge Status	Guidewire Action	Your Action	Type of change made to file	Action taken by Configuration Upgrade Tool
EDIT_DELETE	delete	edit	file changed from base in customer configuration and does not exist in target configuration	<p>If you resolve the file, the Configuration Upgrade Tool takes no action.</p> <p>Double-clicking the file opens your customized file and the original base file in the merge tool specified by <code>upgrader.diff.tool</code> in <code>upgrade.properties</code>. When you close the merge tool, the Configuration Upgrade Tool prompts you to copy the file to the target configuration. If you are sure you want your customized version, you can click Copy prior customized to move the file to the target configuration.</p> <p>The EDIT_DELETE flag appears on a file when your configuration has a customized version of the file but Guidewire has deleted the file from that location. There are two possible reasons for this deletion. One reason is that Guidewire removed the file from BillingCenter. The second reason is that Guidewire has moved the file to a different folder.</p> <p>If Guidewire has completely removed the file, review the <i>BillingCenter New and Changed Guide</i>, release notes, and the Upgrade Diff report for descriptions of the change affecting the deleted file. Then determine if you want to continue moving your customization to the new or changed feature. If not, then the customization will be lost.</p> <p>For the second scenario, find where the file has been moved by searching the target version. Move your customized file to the same location in the working directory and make sure to match any case changes in the filename. When you refresh the list of merge files, the file now appears under the CUSTOMER_EDIT filter. You can now proceed with the merge. If you do not move the file over, you can instead perform the merge manually by opening both files and incorporating the changes.</p>
NO_CHANGE	none	none	file not changed from base configuration in either customer or target configurations	<p>If you resolve the file, the Configuration Upgrade Tool takes no action. The file already exists in the target configuration.</p> <p>Double-clicking opens the file in the text editor specified by <code>upgrader.editor.tool</code> in <code>upgrade.properties</code>. If you make changes, the tool prompts you to copy the file to the target configuration.</p>

Resolved Status Checkboxes

Beneath the change status filters are checkboxes to toggle the visibility of resolved and unresolved files. Use these checkboxes with the change status filters to specify which types of files you want visible in the center panel. For example, you could select **BOTH_EDIT** and **Unresolved** to see files edited in your configuration that have also been updated by Guidewire and are not yet resolved.

The purpose of the resolved status is to have a general idea of the progress you are making in the upgrade. The tool shows the resolved status of the current file (right panel) and the total number of resolved and unresolved files (left panel).

A resolved file is simply a file that you have marked resolved. It does not relate to whether file merging or accepting has occurred.

Bulk Action Buttons

The following buttons in The **Bulk Action** part of the left panel enable you to change the resolved status of a group of selected files:

- **Copy/Mark Resolved**
- **Mark Unresolved**

You can select either one or several files and directories before using these buttons. Use the CTRL key to select multiple files and directories. Selecting a directory selects all files within that directory. You can select all files that match the filters you set by selecting the top-level directory.

After you click **Copy/Mark Resolved**, the Configuration Upgrade Tool opens a dialog detailing the actions it is about to perform.

The tool copies files matching the **CUSTOMER_ADD** and **CUSTOMER_EDIT** filters to the target configuration. If there is already a version of a file in the target configuration, then the tool does not copy the file. A file would be there already if you edited the file and clicked Yes when the tool prompted you to copy the file to the target configuration.

The tool does not do any copying for files matching the **GW_ADD**, **GW_DELETE**, **GW_EDIT**, **NO_CHANGE**, or **Unmergeable** filters. Files matching **GW_ADD**, **GW_EDIT**, **NO_CHANGE**, or **Unmergeable** filters are already present in the target version. Files matching the **GW_DELETE** filter are not in BillingCenter 8.0.4.

You can not bulk resolve multiple files that match the **BOTH_ADD**, **BOTH_EDIT**, or **EDIT_DELETE** filters. Files matching these filters require individual attention. Use the right panel of the Configuration Upgrade Tool to control merging, copying and resolving of these files.

Configuration File Tree

The center panel displays the configuration file tree. Files are color-coded to match filter colors. Files are shown one time, regardless of the number of configurations in which they exist. For information on which configurations a file exists in, select the file and view the right panel. The number of files in each directory that match the selected change status and resolved status filters is shown in parentheses.

File Details Panel

The right panel displays file details for the file you are currently examining, including:

- **Status** – The change status of the file. See “Change Status Filters” on page 111.
- **New base file** – The new version of this file supplied by Guidewire with BillingCenter 8.0.4. If there is not a Guidewire version of this file, such as for **CUSTOMER_ADD**, **EDIT_DELETE** or **GW_DELETE** files, this field is blank.
- **Prior customized file** – The locally customized version of this file from the prior version. If there is not a customized version of this file, such as for **GW_ADD**, **GW_DELETE**, **GW_EDIT** or **NO_CHANGE** files, this field is blank.
- **Prior base file** – The base version of this file in the working directory. If there is not a Guidewire version of this file in the prior base version you are upgrading from, such as for **CUSTOMER_ADD** files, this field is blank.
- **New customized file** – The customized version of this file in the BillingCenter 8.0.4 configuration directory.

The right panel fields are blank if you have multiple files selected.

File Details Panel Actions

The following buttons appear below the file details display in the right panel after you have selected a file:

- **View** – Opens the file in the editor specified in `upgrade.properties`. This button appears for files that are not customized and do not require merging, matching **GW_ADD**, **GW_EDIT**, or **GW_DELETE** filters. Only one of the **View**, **Edit** or **Merge** buttons displays, depending on the file change status.

- **Edit** – Opens the file in the editor specified in `upgrade.properties`. This button appears for custom files that do not require merging, matching the `CUSTOMER_ADD` or `EDIT_DELETE` filters.
- **Merge** – Opens the different versions of the file in the merge tool specified in `upgrade.properties`. This button appears for files that require merging, matching the `BOTH_ADD` or `BOTH_EDIT` filters.
- **Copy prior customized** – Copies the prior customized version of the file to the target configuration. This button is enabled if there is a prior customized version of the file. So it is enabled for files matching `CUSTOMER_ADD`, `CUSTOMER_EDIT`, `BOTH_ADD`, `BOTH_EDIT`, or `EDIT_DELETE` filters.
- **Delete new customized** – Remove the customized version from the target configuration. This reverses the **Copy prior customized** button action. This button is disabled until you have copied a prior customized version of the file into the target configuration.
- **Resolved** – Check this box to label the file resolved. Use the **Resolved** checkbox in the right pane to change the status of a single file. Selecting the **Resolved** checkbox does not copy the file. Use the buttons above this checkbox to handle copying or merging of file versions. You must first unresolve a file before either using the **Delete new customized** action or reapplying changes or merges.

Accepting Files that Do Not Require Merging

The following filters show lists of files that normally do not require merging.

- `CUSTOMER_ADD`
- `CUSTOMER_EDIT`
- `GW_ADD`
- `GW_EDIT`
- `NO_CHANGE`
- `Unmergeable`

You can click the **Copy/Mark Resolved** button in the left panel to resolve groups of these files.

Merging and Accepting Files

Files matching the `BOTH_ADD` and `BOTH_EDIT` filters must be merged before being accepted. Your version must be reconciled with the Guidewire target or base version. In some cases, even if only a single version of the file exists, you might want to look at it before accepting it.

You can use the `pcf.xsd` file in the `modules` directory of the target version to validate merged PCF files.

After you are satisfied with any changes, save the file. This saves the file in a temporary directory. When you close the editor or merge tool, the Configuration Upgrade Tool asks if you want to copy the file to the target configuration. If you click **Yes** (or press `ALT+Y`), the tool copies the file. If you click **No** (`ALT+N`), the tool cancels the popup without copying. The tool always moves files into the target configuration, except if a file is identical to the base or target version. In this case, the tool does not move the file.

Note: Do not edit a file version with `DO_NOT_EDIT` in its file name.

Configuration Merging Guidelines

The first milestone of an upgrade project is to generate the Java and SOAP APIs (by running `gwbc regen-java-api` and `gwbc regen-soap-api`) on the target release. To generate the Java and SOAP APIs, you must:

- Complete the merge of the data model. This includes all files in the /extensions and /fieldvalidators folders.
- Resolve issues encountered while trying to generate the APIs or start the QuickStart application server.

You can generate the Java and SOAP APIs even if you have errors in your enhancements, rules and PCF files.

Typical errors

- **Malformed XML** – The merge tool is not XML-aware. There might be occasions in which the file produced contains malformed XML. To check for well-formed XML, use free third-party tools such as Liquid XML, XML Marker, or Eclipse.
- **Duplicate typecodes** – As part of the merge process, you might have inadvertently merged in duplicate, matching typecodes.
- **Missing symmetric relationship on line-of-business-related typelists** – You might be missing a parent-child relationship with respect to the line-of-business-related typelist, as a result of merging.

After you have generated the Java and SOAP APIs, you can begin the work of upgrading integrations.

Second, after you can successfully generate the Java and SOAP APIs, work on starting the server.

In addition to the typical errors described previously, the server might fail to start due to cyclical graph reference errors. See “Identifying Data Model Issues” on page 135.

You can generate the APIs even if you have errors in your enhancements, rules and PCF files, although error messages will print upon server startup.

After the server can start on the target release, you can begin the database upgrade process.

Continue with the remainder of the configuration upgrade work, which includes evaluating existing PCF files and merging in desired changes.

Data Model Merging Guidelines

From a purely technical standpoint, not addressing the need to incorporate new features, the following are a few guidelines for merging the data model.

Updating Data Types for Case Sensitivity

Data type definitions are case-sensitive in BillingCenter 8.0. If you are upgrading from an early 7.0 version or a version prior to 7.0, you could have column definitions that specify a type using the wrong case. In this event, the server reports an invalid data type error during startup. If the server reports invalid data type errors, check the case of the type attribute for the column in the ETI or ETX extension file for the entity. Extension files are located in the extensions directory of the configuration module. An ETI file exists for custom entity definitions. An ETX file defines extensions to an entity provided with BillingCenter.

Merging Typelists – Overview

There is no automated process to merge typelists. This is a part of the merge process using the Configuration Upgrade Tool. In general, merge typelists before PCF files.

See the *Upgrade Diffs Report* for an inventory of differences in typekeys between the base release and the target release. To retrieve the *Upgrade Diffs Report* follow the procedure described in “Viewing Differences Between Base and Target Releases” on page 97.

Merge in Guidewire-provided typecodes related to lines of business and retire unused typecodes that you merge in. If you do not include these typecodes, you will have errors in any enhancements, rules, or PCF files that reference the typecode. This also simplifies the process for future upgrades as there will be fewer added line of business typecodes to review.

Pay particular attention if any Guidewire-provided typecodes have the same typecode as a custom version. In this case, modify one of the typecodes so they are unique. Contact Guidewire Support for details.

The Configuration Upgrade Tool displays most typelists you have edited in the CUSTOMER_EDIT filter. If your edits are simply additional typecodes, accept your version.

Use Guidewire Studio to verify PCF files, enhancements, and rules to identify any issues with the files and rules that reference typelists.

Merging Typelists – Simple Typelists

Merge in new typecodes from the target version, BillingCenter 8.0.4. If you do not merge the new typecode, you will have errors in any enhancements, rules, or PCF files that reference the typecode. If you do not want to use a new typecode, retire the typecode by setting the `retired` attribute to `true`.

Merging Typelists – Complex Typelists

A typecode can reference typecode values from another typelist using the `<category>` subelement. If a new typecode references an existing typecode, do not merge the new typecode unless the referenced typecode is retired. Otherwise, you are defining a new relationship. If the referenced typecode is also new, merge in both typecodes. If you do not want to use a new typecode, set the `retired` attribute for the typecode to `true`. The following table summarizes how to handle merging new typecodes that reference other typecodes:

Referenced typecode status	Action
new – exists only in target version	Merge in the new typecode and merge in the referenced typecode in its typelist. If you do not want to use the new typecode, retire it by setting the <code>retired</code> attribute of the typecode to <code>true</code> .
active – exists in base or custom version and is not retired	Do not merge the new typecode.
retired – exists in base or custom version and is retired	Merge in the new typecode. If you do not want to use the new typecode, retire it by setting the <code>retired</code> attribute of the typecode to <code>true</code> .

Reviewing Shared Typekey Configuration

As of version 8.0.3, BillingCenter enforces restrictions on the use of shared typekeys among subtypes.

Same Field Name and Typelist with Different Column

In BillingCenter 7.0 and earlier, if a shared typekey had the same field name and typelist, and specified a different column name, BillingCenter created only one of the typekey columns. The shared typekeys were stored in the single column. As of BillingCenter 8.0.3, if a shared typekey with the same field name and typelist specifies a different column name, BillingCenter creates different columns according to the specification. The database upgrade detects shared typekeys using a single column, creates the additional column, and moves the typekey data to the correct column.

Same Field and Column Names with Different Typelists

In BillingCenter 8.0.1 and 8.0.2, a typekey on subtypes could have the same field name and column name and reference different typelists. As of BillingCenter 8.0.3, this configuration is not allowed. The database upgrade reports an error if it detects this condition.

If you have subtypes with typekeys with the same field and column name that reference different typelists, update your data model configuration to use different column names for each typelist. The database upgrade then moves data to the new column to match the updated data model.

Adding State Typelist Extensions to Jurisdiction

BillingCenter versions 7.0 and newer use a Jurisdiction typelist instead of a State typelist. If your environment includes custom extensions to the State typelist, move those extensions to the Jurisdiction typelist.

To move State typelist extensions to the Jurisdiction typelist

1. Open the `modules/configuration/config/extensions/State.ttx` file in your pre-upgrade starting version in a merge tool.
2. Open `modules/configuration/config/extensions/typelist/Jurisdiction.ttx` file in another panel of the merge tool.
3. Merge typecode elements from `State.ttx` to `Jurisdiction.ttx`.

Merging Entity Extensions

BillingCenter 8.0.4 stores extensions in ETI and ETX files. An `Entity.eti` file defines a new entity. An `Entity.etx` file defines extensions to an existing entity.

Correcting File Naming Issues

In BillingCenter 8.0.4, typelist and entity extension files must be named for the typelist or entity. In versions before 8.0, you could have an extension file name such as `Entity_ABC.etx` or `Typelist_ABC.ttx`. As of BillingCenter 8.0, the file root name must be the entity or typelist name or the entity or typelist name followed by a dot. You can use characters after the root name to include custom name components. For example, `Entity.ABC.etx` is a valid entity extension file name. `Typelist.ABC.ttx` is a valid typelist extension file name. If you have extension files that have names that include characters other than the entity name, rename the files to put the extra characters after a dot.

Correcting Data Type References

BillingCenter entity files must use case-sensitive references to data types. For example, setting a `<column-override>` to have `type="shorttext"` is not the same as setting `type="ShortText"`. In this case, the former is valid while the latter is not.

Review each entity extension you have added to make sure data type references are set with the correct case.

To review and correct extension data type references

1. In Studio, expand `configuration` → `config` → `Extensions` → `Entity`.
2. Double-click each ETX file. If the file has an invalid data type reference, Studio reports that the extension field overrides validator detected a column override that refers to a non-existent data type.
3. For any such errors, select the column. Then select the correct case-sensitive `Value` for the `type` from the drop-down list.

Reviewing Optional Indexes

Guidewire often adds indexes to entities in the target configuration to improve the performance of database queries in BillingCenter 8.0.4. BillingCenter requires some of these indexes. Guidewire adds required indexes to entity definitions in the data model. Other indexes are recommended for most installations but can be disabled if they negatively impact performance. Guidewire adds optional indexes to entity extensions so you can disable any of these indexes if necessary.

Use the Configuration Upgrade Tool to resolve extension files. When you merge your custom extensions with Guidewire changes, review each new index added by Guidewire. In most cases, include the new index in the merged extension file. You can modify or remove index definitions based on usage in your deployment.

Updating `setterScriptability` Attributes

The `setterScriptability` attribute can no longer be set to `external` as of BillingCenter 8.0. For any instances you have of the attribute `setterScriptability` set to `external`, change the value to `all`.

Reviewing Custom Extensions

Generate and review the data dictionary for the target version to identify any custom extensions that are now obsolete due to Guidewire adding a similar field to the base BillingCenter.

To generate the data dictionary

1. From the command line, navigate to the `bin` directory of the target version.
2. Run the command `gwbc regen-dictionary`.

This command generates the data and security dictionaries in the `build/dictionary` directory of the target version. To view the data dictionary, open `build/dictionary/data/index.html` in a web browser.

Compare the target version data dictionary with the version in your current environment. If you have extensions that are now available as base fields, consider migrating the data in those fields to the base version. Consider whether an extension is still on the appropriate entity. A new entity could be a more appropriate location for the extension. Review key data model changes that might impact your custom extensions.

If you change an extension location or migrate to a new base field, update any PCF, rule or library that references the extension to reference the new location.

Reconciling the Database with Custom Extensions

Extensions defined in ETI and ETX files must match the physical database. Delete all physical columns in the database that are not part of the base installation or defined as extensions before starting the server.

Removing Obsolete Attributes

Guidewire has removed the `deletefk` and `onDelete` attributes of the `<foreignkey>` and `<edgeForeignKey>` elements. These attributes were deprecated in an earlier major version. Now that the attributes are removed from the schema for entity definition files, if the attributes are listed, the server reports an error and does not start. Remove any occurrences of `deletefk` and `onDelete` attributes from `<foreignkey>` and `<edgeForeignKey>` elements in custom entities.

Updating Extractable Edge Foreign Keys

Guidewire has removed the `<implementsEntity>` element from `<edgeForeignKey>` and `<edgeForeignKey-override>`. In BillingCenter 8.0, to make an edge foreign key extractable, set the Boolean `extractable` attribute on the element to `true`.

For any extractable edge foreign keys and edge foreign key overrides, delete the `<implementsEntity>` element from the key definition. Then add the attribute `extractable="true"` to the `<edgeForeignKey>` or `<edgeForeignKey-override>` element.

Converting Money to MonetaryAmount

BillingCenter upgrade cannot automatically convert the `Money` data type to the `MonetaryAmount` data type. If you created entity extensions, the upgrade process will not upgrade your extensions that include properties that use the `Money` data type.

Before you upgrade, manually update any extension properties that use the `Money` data type.

Define the `MonetaryAmount` property as follows:

- The name of the new `MonetaryAmount` property is the same as the name of the `Money` property
- If the old `Money` property had a `columnName` attribute defined as something other than the `Money` property name, use that old `Money.columnName` as the name of the new `MonetaryAmount.amountColumnName` attribute.
- For extensions to entities that will be `InCurrencySilo` entities in BillingCenter 8, you must add a `DefaultValueZero` tag if your old `Money` property had the `default` attribute set to 0.
- Set `scaleToCurrency` to `true` unless you have a requirement to do otherwise.
- Set the `soapNullOk` attribute to `true`

If you used an extension column to represent money, but did not set the column to the `money` datatype, contact Guidewire Support.

The following examples show how you must redefine `Money` properties in your extensions to `MonetaryAmount` properties before you proceed with upgrade:

Old Total:

```
<column
  name="Total"
  type="money"/>
```

New Total:

```
<monetaryamount
  name="Total"
  amountColumnName="Total"
  soapNullOk="true" />
```

Old Total with default = 0:

```
<column
  name="Total"
  default="0"
  type="money"/>
```

New Total:

```
<monetaryamount
  name="Total"
  amountColumnName="Total"
  soapNullOk="true" /
  <tag name="DefaultValueZero"/>
</monetaryamount>
```

Old Total where name and columnName differ:

```
<column
  name=" Total"
  columnName="totalColumn"
  type="money"/>
```

New Total:

```
<monetaryamount
  name="Total"
  amountColumnName="totalColumn"
```

```
soapNullOk="true" />
```

Changes to the Logging API

Guidewire updated the logging API between BillingCenter 7.0.1 and 7.0.2. Although changes to logging infrastructure were extensive, the purpose of these changes is simplification of logging usage. This document describes changes to the Guidewire logging API. If you are upgrading from a version prior to BillingCenter 7.0.2, use this section as a guide to update your configuration files to the new logging API.

Conceptual Changes to Logging

Old API

<code>com.guidewire.logging.Logger</code>	The Logger class implements all logging functions. Instantiate the class with a new statement. This class is a wrapper around the Log4J Logger class.
<code>com.guidewire.logging.LoggerFactory</code>	The LoggerFactory class has two purposes. First, the class instantiates the logging infrastructure and determines the logging configuration. Second, the class is a factory that produces Logger instances.
<code>com.guidewire.logging.LoggerCategory</code>	The LoggerCategory class is a subclass of the Logger class. An instance of the LoggerCategory class behaves exactly the same way as instance of Logger, but LoggerCategory also maintains a set of static members, which are predefined loggers.
<code>com.guidewire.xx.system.logging.XXLoggerCategory</code> in which xx is a product-specific code such as bc, cc, or pc.	The XXLoggerCategory classes are application-specific subclasses of LoggerCategory. Normally, application-specific LoggerCategory classes maintain additional static Logger members for applications to use.

New API

<code>gw.pl.logging.Logger</code>	<p>Logger was converted from a class to an interface in BillingCenter 7.0.2. In 8.0, Logger is deprecated. You can update code to use <code>org.slf4j.Logger</code> instead of <code>gw.pl.logging.Logger</code>.</p> <p>The Logger interface provides all necessary functionality and hides implementation. This Logger interface explicitly prohibits certain functions that the previous Logger class allowed:</p> <ul style="list-style-type: none"> • You cannot set logging level within the application • You cannot add nor remove appenders within the application <p>The purpose of the Logger interface is to log application-specific messages. Every application component must use its own Logger instance to log messages relevant to the component itself.</p> <p>Do not perform logger management from within the component, such as defining the logging level for a logger. Instead, use the <code>logging.properties</code> file and the application interface to control logging levels and appenders.</p>
<code>gw.pl.logging.LoggerFactory</code>	<p>The LoggerFactory class retains its original functionality, but some methods have changed.</p> <p>This LoggerFactory has two purposes. First, the class instantiates the logging infrastructure and determines the logging configuration. Second, the class is a factory that produces Logger instances.</p>
<code>gw.api.util.Logger.forCategory</code>	<p>The forCategory method of the Logger class returns a Logger for the category, which is passed as a parameter to the forCategory method.</p>
<code>com.guidewire.xx.system.logging.XXLoggerCategory</code> in which xx is a product-specific code such as bc, cc, or pc.	<p>The XXLoggerCategory classes are application-specific subclasses of LoggerCategory. They retain their function of maintaining additional static Logger members for applications to use, but the static members now are instances of the Logger interface. You can no longer instantiate application-specific subclasses of LoggerCategory.</p>
<code>gw.api.system.XXLoggerCategory</code> in which xx is a product-specific code such as bc, cc, or pc.	<p>A mirror class to expose the logger category. It inherits all loggers defined in its <code>gw.pl</code> counterpart.</p>

Instantiating Loggers

Old API

With the old API, you can instantiate logger instances using `Logger`, `LoggerCategory`, `LoggerFactory` or an instance of `LoggerCategory`. Any of the following statements instantiates a logger:

```
Logger logger1 = new Logger("Logger1");
Logger logger12 = new Logger(logger1, "Sublogger2");
LoggerCategory category1 = new LoggerCategory("Category1");
LoggerCategory category12 = new LoggerCategory(category1, "Subcategory2");
Logger factoryLogger1 = LoggerFactory.getInstance().getLogger("FactoryLogger1");
Logger factoryLogger12 = LoggerFactory.getInstance().getLogger(factoryLogger1,"Sublogger2");
Logger apiLogger = LoggerCategory.API;
LoggerCategory apiCategory = LoggerCategory.API;
Logger apiSubLogger = new LoggerCategory(LoggerCategory.API, "WebAPI");
LoggerCategory apiSubCategory = new LoggerCategory(LoggerCategory.API, "WebAPI");
```

New API

With the new API, you work only with instances of the `Logger` interface. You can no longer directly instantiate logger instances, so the new API supports only a few methods to obtain a logger instance:

```
// Using gw.* package
import gw.util.*;
import gw.api.system.*;
import gw.api.util.*;

ILogger apiLogger = PLLoggerCategory.API;
ILogger apiSubLogger = Logger.forCategory(PLLoggerCategory.API, "WebAPI");

// Using com.guidewire.* package
import com.guidewire.logging.*;
Logger logger1 = LoggerFactory.getLogger("Logger1");
Logger logger12 = LoggerFactory.getLogger(logger1, "Sublogger2");
Logger apiLogger = LoggerCategory.API;
Logger apiSubLogger = LoggerFactory.getLogger(PLLoggerCategory.API, "WebAPI");
```

The new API loses no functionality compared with the old API, but fewer arbitrary options exist.

Note: The `LoggerFactory` class no longer has a `getInstance()` method. The `LoggerFactory.getLogger()` method is now static.

Logging Messages

After you obtain an instance of the `Logger` with the new API, you can use the same methods as the old API to log messages. However, a new interface also allows SLF4J formatting of the messages.

Old API

```
logger.info("Started application " + appName + " with parameters " + parms.toString());
logger.info("Listening to the port " + Integer.toString(portNumber));
```

New API

```
if (wantOldStyle) {    // Old style
    logger.info("Started application " + appName + " with parameters " + parms.toString());
    logger.info("Listening to the port " + Integer.toString(portNumber));
} else {                // New style
    logger.info("Started application {} with parameters {}", appName, parms.toString());
    logger.info("Listening to the port {}", new Integer(portNumber));
}
```

Passing Loggers as Parameters

With the new API, the `LoggerCategory` class exists and has static members. However, those members are instances of the `Logger` interface instead of the `LoggerCategory` itself.

Old API

```
private LoggerCategory getApiLogger() {
    return LoggerCategory.API;
}

// ...
LoggerCategory myLogger = getApiLogger();
myLogger.debug("...");
```

New API

```
// Using gw.* package.
private gw.pl.logging.Logger getApiLogger() {
    return PLLoggerCategory.API;
}
// ...
gw.pl.logging.Logger myLogger = getApiLogger();
myLogger.debug("...");

// Using SLF4J.
private org.slf4j.Logger getApiLogger() {
```

```

        return PLLoggerCategory.API;
    }

    // ...
    org.slf4j.Logger myLogger = getApiLogger();
    myLogger.debug("I am Logger {}", myLogger.toString());
}

```

Adding DDL Configuration Options to database-config.xml

The configuration upgrade includes an automated step to move the database configuration from `config.xml` to `database-config.xml`. The automated step transforms most of the configuration to the 8.0 standard. However, the automated step does not transform certain DDL-related configuration settings. If you have DDL-related configuration settings for compression, Oracle SecureFile LOBs, or partitioning, recreate the configuration in the `database-config.xml` file.

DDL configuration setting changes only apply to new objects. For example, if you change an existing table from BasicFile to SecureFile LOBs, only new LOB columns will be SecureFile LOBs.

For instructions, see the following topics:

- “Configuring Compression” on page 24 in the *Installation Guide*
- “Configuring BillingCenter to Use Oracle SecureFile LOBs” on page 31 in the *Installation Guide*
- “Configuring Table Partitioning for Oracle” on page 32 in the *Installation Guide*

Merging Changes to Field Validators

The `<ValidatorDef>` element in `fieldvalidators.xml` accepts new attributes with BillingCenter 8.0. All of the new attributes are optional. These attributes, the values that you can set the attributes to, and the default value of the attributes are listed in the following table:

Attribute	Values	Default	Description
validation-level	none	strict	The validation-level is passed to the Gosu validators. The functionality for each validation level is specific to the custom validator.
	relaxed		
	strict		
validation-type	gosu	regex	If validation-type is set to regex, the value of the <code><ValidatorDef></code> defines a regular expression that BillingCenter uses to validate data entered into a field that uses the field validator.
	regex		If the validation-type is set to gosu, the value of the <code><ValidatorDef></code> is a Gosu class. The Gosu class must extend <code>FieldValidatorBase</code> and override the <code>validate</code> method. See <code>gw.api.validation.PhoneValidator</code> for an example. Ensure that any Gosu validators that you define are low-latency for performance reasons.

Guidewire has updated some `<ValidatorDef>` elements in `fieldvalidators.xml` to use the new attributes. For `<ValidatorDef>` elements that you have customized, review the use of the new attribute to see if the behavior is what you want. For `<ValidatorDef>` elements that you have not customized, you can accept the new attributes.

Renaming PCF files According to Their Modes

In BillingCenter 8.0, a PCF file may contain only a single mode, and must include the name of its mode, if any, in the file name. Violations of this rule produce compilation errors in Guidewire Studio. For example, if a file `MyFileDV.pcf` had previously defined two modes, abc and xyz, those modes must now be split into separate files, named `MyFileDV.abc.pcf` and `MyFileDV.xyz.pcf`. Even if a PCF file only contains a single mode, but that mode is not included in the file name, you must still rename the file to include the mode.

Guidewire has renamed all PCF files included in the default 8.0 configuration. However, the Configuration Upgrade Tool might not automatically fix some of your own added or changed files. In particular, take notice of `EDIT_DELETE` conflicts during the three-way merge process. Guidewire could have renamed or split apart the file based on its PCF modes rather than deleted the file. In that case, the new PCF file or files are likely to be in the same directory. Merge your changes into the new file or files.

Updating Rounding Mode Parameter

BillingCenter 8.0.4 has changed from using the `RoundingMode` parameter to the `DefaultRoundingMode` parameter.

Check that you either have no usages of `DefaultRoundingMode` in your configuration, including Gosu code and PCF files, or that if used, it has the same value as `RoundingMode`.

Then, update all usages of `RoundingMode` to use `DefaultRoundingMode`. Define the `DefaultRoundingMode` parameter in `config.xml` to reflect the previous `RoundingMode` parameter value. Change the `RoundingMode` param element in `config.xml` by changing the name to `DefaultRoundingMode` and removing `ROUND_` from the front of the value

For example, change:

```
<param name="RoundingMode" value="ROUND_HALF_EVEN"/>
```

to:

```
<param name="DefaultRoundingMode" value="HALF_EVEN"/>
```

Finally, before the database upgrade, run the following SQL:

```
DELETE FROM bc_systemparameter WHERE name = 'config_param_DefaultRoundingMode'
```

Merging Display Properties

The Configuration Upgrade Tool updates display properties files, such as `display.properties`, as described in “Upgrading Display Keys” on page 108 to create a merged file with the extension `.merged`. You could have conflicts in the files if you have a different number of parameters for a key than the 8.0 version or if you have a different value.

If the number of parameters differs from the 8.0 version, match your parameter set to the 8.0 version of the key.

If the value is different, choose which value you want to use in your BillingCenter configuration.

Merge changes into `display.properties.merged`. When you save the file, the Configuration Upgrade Tool saves it to the configuration module without the `.merged` extension.

If you have added locales, you can export a full list of display keys and typelists from the default BillingCenter 8.0.4 locale to any locale you have defined. This list includes a section for display keys and typelists that do not yet have values defined for your locale. You can use this list to determine which display keys and typelists require localized values. You can then specify those values and import the list. See “Translating New Display Properties and Typecodes” on page 129.

In BillingCenter 8.0, Studio trims trailing spaces from display keys by default. You can modify this behavior using the following procedure:

1. Click File → Settings.
2. Under IDE Settings click Editor.
3. Under Other, change the value of Strip trailing spaces on Save to None.
4. Click OK.

Merging Other Files

In some cases, you cannot effectively merge the differences between files using a comparison tool. In particular, `config.xml`, `logging.properties`, and `scheduler-config.xml` often have many changes between major versions. Consider adding your custom changes to the new Guidewire-provided version instead of merging from prior versions if the presentation of these files in the merge tool is too daunting.

During startup, BillingCenter 8.0.4 reports a warning message if you have configuration parameters defined in `config.xml` that BillingCenter 8.0.4 does not use. BillingCenter ignores any unused parameters. You might have old parameters in `config.xml` that BillingCenter does not use. If BillingCenter 8.0.4 reports that there are unknown parameters specified, remove these parameters from `config.xml`.

If your installation contains a language that is not one of the core Guidewire-supported languages in the base configuration, in `config.xml` copy the value of `DefaultApplicationLocale` to `DefaultApplicationLanguage`. The core Guidewire-supported languages in the base configuration are U.S. English, Italian, German, Spanish, French, Chinese, and Japanese.

Fixing Gosu Issues

Review additions and changes to Gosu code in the BillingCenter New and Changed Guide. Update your Gosu code for these changes. Use the procedures in this topic to detect and fix these issues.

See also

- “New and Changed in Gosu in 8.0” on page 47 in the *New and Changed Guide*
- “What’s New and Changed in 8.0 Maintenance Releases” on page 17 in the *New and Changed Guide*

Gosu Case Sensitivity

BillingCenter 8.0 has strict case-sensitivity for Gosu code.

To detect and fix case-mismatch issues

1. Right-click a folder in the Project pane and select Analyze → Run Inspection by Name....

Note: Do not select the whole project as the inspection is resource intensive.

2. Enter case and double-click Name is referenced with improper case.
3. In the dialog, set Inspection scope to Directory.

4. Deselect **Include test sources**.
5. Click **OK**.
6. In the **Results** pane, expand **Case mismatch issues**, if present.
7. Right-click the **Name is referenced with improper case** issue type, and click **Apply Fix 'Case mismatch issues'**.
8. Click the **Save All** icon.
9. Repeat this procedure for the selected folder until no case mismatch issues are reported or the count stops dropping. It might not drop all the way to zero. Keep a record of any folders that do not reach zero errors.
10. Continue this process for all folders containing files with Gosu code.
11. If any folders have an error count above zero, and the count is not dropping after you apply the fix, compile the project to detect other errors.

Inequality Operator

The inequality operator `<>` is no longer valid and must be replaced with `!=`.

To detect and fix the obsolete inequality operator

1. Right-click a folder in the **Project** pane and select **Analyze → Run Inspection by Name....**

Note: Do not select the whole project as the inspection is resource intensive.
2. Enter `The <>` and double-click `The <> operator is obsolete`.
3. In the dialog, set **Inspection scope** set to **Directory**.
4. Click **OK**.
5. In the **Results** pane, expand **Equality issues**, if present.
6. Right-click issue type `The <> operator is obsolete`, and click **Apply Fix 'Equality issues'**.
7. Click the **Save All** icon.
8. Repeat this procedure for the selected folder until no equality issues are reported or the count stops dropping. It might not drop all the way to zero. Keep a record of any folders that do not reach zero errors.
9. Continue this process for all folders containing files with Gosu code.
10. If any folders have an error count above zero, and the count is not dropping after you apply the fix, compile the project to detect other errors.

Ambiguous Method Calls

Previous versions of Gosu reported a warning on ambiguous method calls. Ambiguous method calls can hide a logical bug in your code. Previously, the Gosu compiler selected the best matching method to remove ambiguity. For BillingCenter 8.0, ambiguous calls are now an error instead of a warning. Studio now has a code inspection to identify and optionally fix any ambiguous code to previous Studio behavior. This inspection is disabled by default. To find and fix potential logical errors, Guidewire recommends that you run the inspection and carefully individually analyze every ambiguous call before applying any proposed fix.

To detect and fix ambiguous method calls

1. Right-click a folder in the **Project** pane and select **Analyze → Run Inspection by Name....**

Note: Do not select the whole project as the inspection is resource intensive.
2. Enter `The method` and double-click `The method call is ambiguous, it can be fixed by adding casts`.

3. In the dialog, set **Inspection scope** to **Directory**.
4. Deselect **Include test sources**.
5. Click **OK**.
6. In the **Results** pane, expand **The method call is ambiguous, it can be fixed by adding casts**, if present.
7. Analyze and fix any ambiguous method calls that are reported.
8. Repeat this procedure for the selected folder until no ambiguous method call issues are reported or the count stops dropping. It might not drop all the way to zero. Keep a record of any folders that do not reach zero errors.
9. Continue this process for all folders containing files with Gosu code.
10. If any folders have an error count above zero, and the count is not dropping after you apply the fix, compile the project to detect other errors.

Nested Comments

Gosu supports nested comments. The purpose of nested comments is to quickly comment out large swaths of code temporarily while avoiding compiler errors whenever the enclosed code contains comments.

In earlier releases, the Gosu compiler searched only for “`/* /`” after encountering a comment that opened with “`/*`”. This behavior permitted developers to include dividing lines within lengthy comments, like the following example.

```
////////////////////////////////////////////////////////////////////////
```

In BillingCenter 8.0.4, the Gosu compiler searches for “`/* */`” after encountering a comment that opens with “`/*`” in case the comment body contains a nested comment. Because the comment line in the preceding example begins with “`/*`”, the compiler begins searching for the close of the nested comment and never finds one.

Following an upgrade to BillingCenter 8.0.4, the Gosu compiler may produce the following error message:

```
unclosed comment
This occurs in multiple-line comments that use the open and close comment marks "/*" and "*/" if the
comment body contains the character sequence "/*".
```

To resolve unclosed comment errors

1. If the Gosu compiler reports the unclosed comment error, open the source file in Studio.
2. Rewrite any comments that inadvertently include the character sequence “`/*`” within the body of comments. In the preceding example, you could avoid the problem by inserting a space between the slash and the asterisk or by changing to a sequence of characters other than asterisks.
If there are a number of errors for one source file, consider opening the source in the pre-upgrade version of Studio. Then you can compare the commented sections between the old and new Gosu behavior.
3. Compile the project to find any further errors.

Upgrading Rules to BillingCenter 8.0.4

The Configuration Upgrade Tool does not upgrade rules. The tool classifies rules in the unmergeable filter. Within the target directory, Guidewire-provided default rules are located in `modules/configuration/config/rules`. The Configuration Upgrade Tool moves your custom rules to `modules/configuration/config/rules`.

Guidewire also copies the default rules for the current release to a BillingCenter 8.0.4 Rules folder within `modules/configuration/config/rules`. Use Studio to update your rules. You can use the rules in the BillingCenter 8.0.4 folder as a comparison. Compare your custom rules to the new default 8.0.4 versions and update your rules as needed.

You might find it useful to do a bulk comparison of default rules from the base release against the 8.0.4 versions to determine what types of changes Guidewire has made.

To compare rules between versions using the Rule Repository Report

1. If you want to compare default rules only, temporarily remove custom rules from your starting version by moving the `modules/configuration/config/rules` directory to a location outside the BillingCenter directory.

If you want to compare your custom rules against the BillingCenter 8.0.4 rules, do not move the `modules/configuration/config/rules` directory from your starting version. However, do remove the `BillingCenter<base version>` directory from `modules/configuration/config/rules/rules` of the starting version if this directory exists.

2. Open a command window.
3. Navigate to the `bin` directory of your starting version.
4. Enter the following command:

```
gwbc regen-rulereport
```

This command creates a rule repository report XML file in `build/rules`.

5. Append the starting version number to the XML file name.
6. Restore moved directories to the starting version.
7. Install files for a fresh BillingCenter 8.0.4 version. This is a separate configuration from the target configuration that you have merged. This version will only contain the default rules provided with BillingCenter 8.0.4.
8. Navigate to the `bin` directory of the new BillingCenter 8.0.4 version.

9. Enter the following command:

```
gwbc regen-rulereport
```

This command creates a rule repository report XML file in `build/rules`. There is a slight change to the path between the versions.

10. Append the target version number to the XML file name.

11. Open both rule report XML files in a merge tool. You do not merge base rules using the rule repository reports. However, looking at changes that Guidewire has made to the base rules can help you determine the types of changes you must make in your custom rules.

In your merge tool, disable whitespace differences and comments to reduce the amount of inconsequential differences shown between rules.

Update custom rules using Studio. Studio does not compare your rules directly with target rules. However, Studio provides powerful Gosu editing features not available in a standard text editor that can alert you to issues.

In Studio, you can compare custom rules to default BillingCenter 8.0.4 rules by opening the default rules in the BillingCenter 8.0.4 directory within `configuration → config → Rule Sets`. When you have finished updating all of your custom rules, delete the BillingCenter 8.0.4 rules directory from `modules/configuration/config/rules`.

The BillingCenter 8.0.4 default rules are enabled because some features depend on these rules.

Translating New Display Properties and Typecodes

BillingCenter 8.0.4 adds new display properties and typecodes. If you have defined additional locales, export these new display properties and typecodes to a file, define localized values, and reimport the localized values. If you do not have additional locales defined in your BillingCenter environment, skip this procedure.

To localize new display properties and typecodes

1. Export display keys by running the following command from your BillingCenter 8.0.4 environment `BillingCenter/bin` directory:

```
gwbc export-l10ns -Dexport.file="translation_file" -Dexport.locale="language to export"
```

2. Open the exported translation file in a text editor. The first section of the file lists display properties and typecodes that have a localized value. The second section lists display properties and typecodes that do not have a localized value.

3. Specify localized values for the untranslated properties.

4. Save the updated file.

5. Import the updated file by running the following command from your BillingCenter 8.0.4 environment `BillingCenter/bin` directory:

```
gwbc import-l10ns -Dimport.file="translation_file" -Dimport.locale="language to import"
```

After you import the localized typecodes and display keys, you can view them in Studio.

Validating the BillingCenter 8.0.4 Configuration

This topic includes procedures to validate the upgraded configuration.

Using Studio to Verify Files

You can use Studio to verify classes and enhancements, including libraries, PCF files, rules, and typelists without having to start BillingCenter. Do not start BillingCenter at this point. Studio can run without connecting to the application server.

To validate Studio resources

1. Start Guidewire Studio by running `gwbc studio` from the `BillingCenter\bin` directory.
2. Click **Analyze** → **Inspect Code....**
3. Set the **Inspection scope** to **module 'configuration'**.
4. Click **OK**. Studio runs inspections to identify incorrect Gosu syntax, issuing either a warning or an error.
5. Correct all identified errors with Studio. You can defer fixing warnings.

Starting BillingCenter and Resolving Errors

IMPORTANT In the process described in this section, do not point the BillingCenter server at a production database. The goal of this process is to test the configuration upgrade. Create an empty database account and point BillingCenter to this account for this process. See “Configuring the Database” on page 23 in the *Installation Guide* and “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.

Upon starting the server for the first time, you might receive errors that prevent the server from starting. In general, fixing errors and starting the server is an iterative process that involves:

1. Start the server for the target configuration.
BillingCenter encounters a configuration error and shuts down.
2. Copy the error message to a log file.

3. Locate the configuration causing the error, such as a line of code in a PCF.

4. Comment out the offending line.

After the server starts successfully, look at the log and start solving errors and introducing solutions into the environment. Assign errors to developers on your team.

5. Copy the commented file to the test bed for later analysis.

6. Begin again with step 1. Continue until the server starts successfully.

When the server starts successfully, resolve any remaining issues in the configuration that caused startup errors. Attempt to resolve each error individually and start the server to see if the fix worked.

Building and Deploying BillingCenter 8.0.4

After you apply and validate an upgrade to the configuration environment, rebuild and redeploy BillingCenter. Before you begin, make sure you have carefully prepared for this step. In particular, make sure you have updated your infrastructure appropriately.

Review this topic and then rebuild and redeploy BillingCenter to the application server. See “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide* of the target version for instructions.

WARNING Do not yet start BillingCenter. Only package the application file and deploy it to the application server. Starting BillingCenter begins the database upgrade.

If you have multiple Guidewire products, then upgrade, build, and deploy each individually before attempting to reintegrate them.

The Build Environment

With the exception of the database configuration, the first time you start the application server use the unmodified `config.xml` and `logging.properties` files provided with the target configuration. After the server starts successfully, you can merge in specific configurations of these files.

If you encounter build failures due to data dictionary generation, you can comment out this dictionary generation. Then, as you start the server, it reports any PCF configuration errors. After you have corrected PCF configurations, un-comment the dictionary generation and rebuild the application file.

Preserving JAR Files

Place custom JAR files in the `/config/lib` directory. Building and deploying a WAR or EAR file copies the JAR file into the appropriate place for it to be accessed by BillingCenter. JAR files in this location survive the upgrade process.

Upgrading the BillingCenter 7.0.x Database

This topic provides instructions for upgrading the BillingCenter database to BillingCenter 8.0.4.

If you are upgrading from a 3.0.x version, see “Upgrading the BillingCenter 3.0.x Database” on page 249 instead.

This topic includes:

- “Upgrading Administration Data for Testing” on page 134
- “Identifying Data Model Issues” on page 135
- “Verifying Batch Process and Work Queue Completion” on page 136
- “Purging Data Prior to Upgrade” on page 136
- “Validating the Database Schema” on page 137
- “Checking Database Consistency” on page 138
- “Creating a Data Distribution Report” on page 138
- “Generating Database Statistics” on page 139
- “Creating a Database Backup” on page 140
- “Updating Database Infrastructure” on page 140
- “Preparing the Database for Upgrade” on page 140
- “Setting Linguistic Search Collation” on page 141
- “Field Encryption and the Upgraded Database” on page 142
- “Customizing the Upgrade” on page 142
- “Running the Commission Payable Calculations Process” on page 153
- “Configuring the Database Upgrade” on page 153
- “Checking the Database Before Upgrade” on page 160
- “Disabling the Scheduler” on page 160

- “Suspending Message Destinations” on page 161
- “Starting the Server to Begin Automatic Database Upgrade” on page 161
- “Viewing Detailed Database Upgrade Information” on page 182
- “Dropping Unused Columns on Oracle” on page 183
- “Exporting Administration Data for Testing” on page 184
- “Upgrading Phone Numbers” on page 185
- “Final Steps After The Database Upgrade is Complete” on page 186

Upgrading Administration Data for Testing

You might want to create an upgraded administration data set for development and testing of rules and libraries with BillingCenter 8.0.4. You can wait until the full database upgrade is complete and then export the administration data, as described in “Exporting Administration Data for Testing” on page 184. Or, you can upgrade only the administration data to have this data available earlier in the upgrade process. Use the procedure in this section to create an upgraded administration data set before upgrading the full database.

To upgrade administration data

1. Export administration data from your current (pre-upgrade) BillingCenter production instance:
 - a. Log on to BillingCenter as a user with the `viewadmin` and `soapadmin` permissions.
 - b. Click the **Administration** tab.
 - c. Choose **Import/Export Data**.
 - d. Select the **Export** tab.
 - e. Select **Admin** from the **Data to Export** dropdown.
 - f. Click **Export**. BillingCenter exports an `admin.xml` file.
2. On a new pre-upgrade development environment based on your production configuration, create an empty version of `importfiles.txt` in the `modules/configuration/config/import/gen` directory.
3. Create empty versions of the following CSV files:
 - `activity-patterns.csv`
 - `authority-limits.csv`
 - `reportgroups.csv`
 - `roleprivileges.csv`
 - `rolereportprivileges.csv`Leave `roles.csv` as the original complete file.
4. Start the development environment server by opening a command prompt to `BillingCenter/bin` and entering the following command:
`gwbc dev-start`
5. Import this administration data into the development environment.
 - a. Log on to BillingCenter as a user with the `viewadmin` and `soapadmin` permissions.
 - b. Click the **Administration** tab.
 - c. Choose **Import/Export Data**.
 - d. Select the **Import** tab.
 - e. Click **Browse....**

- f. Select the `admin.xml` file that you exported in step 1.
 - g. Click **Open**.
6. Create a backup of the new development environment database.
7. Create a new database account for the development environment on a database management system supported by BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <http://guidewire.custhelp.com>.
See “Configuring the Database” on page 23 in the *Installation Guide* for instructions to configure the database account.
8. Restore the backup of the database containing the imported administration data into the new database.
9. Connect your upgraded target BillingCenter 8.0.4 configuration to the restored database.
10. Start the BillingCenter 8.0.4 server to upgrade the database.
11. Export the upgraded administration data:
 - a. Start the BillingCenter 8.0.4 server by navigating to `BillingCenter/bin` and running the following command:
`gwbc dev-start`
 - b. Open a browser to BillingCenter 8.0.4.
 - c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
 - d. Click the **Administration** tab.
 - e. Choose **Import/Export Data**.
 - f. Select the **Export** tab.
 - g. For **Data to Export**, select **Admin**.
 - h. Click **Export**. Your browser will note that you are opening a file and will prompt you to save or download the file.
 - i. Select to download the `admin.xml` file. You can import this XML file into local development environments of BillingCenter 8.0.4.

Identifying Data Model Issues

Before you upgrade a production database, identify issues with the datamodel by running the database upgrade on an empty database. This process does not identify all possible issues. The database upgrade does not detect issues caused by specific data in your production database. Instead, this procedure identifies issues with the data model.

Complete the following procedure to identify data model issues, and correct any issues on an empty schema. Then, follow the full list of procedures in this topic to upgrade a production database. This list begins with “Verifying Batch Process and Work Queue Completion” on page 136 and finishes with “Final Steps After The Database Upgrade is Complete” on page 186.

To identify data model issues

1. Create an empty schema of your starting version database. You can do this in a development environment by pointing the development BillingCenter installation at an empty schema and starting the BillingCenter server. See “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.

2. Complete the configuration upgrade for data model files in your starting version, according to the instructions in “Upgrading the BillingCenter 7.0.x Configuration” on page 95. You do not need to complete the merge process for all files.
3. Configure your upgraded development environment to point to the database account containing the empty schema of your old version. See “Configuring a Database Connection” on page 62 in the *Installation Guide*.
4. Start the BillingCenter server in your upgraded development environment. The server performs the database upgrade to BillingCenter 8.0.4. See “Starting the Server to Begin Automatic Database Upgrade” on page 161.
5. Check for errors reported during the upgrade process. Resolve any issues before upgrading your production database. You can use the `IDatabaseUpgrade` plugin to run custom SQL before and after the database upgrade. For more information, see “Running Custom Version Checks and Triggers” on page 143.

Verifying Batch Process and Work Queue Completion

All batch processes and work queues must complete before beginning the upgrade. Check the status of batch processes and work queues in your current production environment.

To check the status of batch processes and work queues

1. Log in to BillingCenter as the superuser.
2. Press `Alt + Shift + T`. BillingCenter displays the **Server Tools** tab.
3. Click **Batch Process Info**.
4. Select **Any** from the **Processes** drop-down filter.
5. Click **Refresh**.
6. Check the **Status** column for each batch process listed. This list also includes batch processes that are writers for distributed work queues. If any of the batch processes have a **Status** of **Active**, wait for the batch process to complete before continuing with the upgrade.

Purging Data Prior to Upgrade

This topic includes recommendations for purging certain types of data from the database prior to upgrade. Removing unused records can improve the performance of the database upgrade and BillingCenter.

Purging Old Messages from the Database

Purge completed inactive messages before upgrading the database. Doing so reduces the complexity of the database upgrade.

You can use the following command from the current (pre-upgrade) customer configuration `admin/bin` directory to purge completed messages from the `bc_MessageHistory` table:

```
messaging_tools -password password -server http://server:port/instance -purge MM/DD/YYYY
```

This tool deletes completed messages with a send time before the date `MM/DD/YYYY`.

Or, you can use the following web service API:

```
IMessageToolsAPI.purgeCompletedMessages(java.util.Calendar cutoff)
```

Periodically purge old messages to prevent the database from growing unnecessarily.

Purge messages from the database before starting BillingCenter, so the database upgrade does not attempt to convert those rows.

You cannot resend old messages after the upgrade. This is because integrations change and the message payload might be different. It is important that messages that have failed or not yet been consumed finish prior to upgrading.

After you purge completed inactive messages, reorganize the `bc_MessageHistory` table. You might also want to rebuild any indexes on the table. Contact Guidewire Support if you need assistance.

Purging Completed Workflows and Workflow Logs

Each time BillingCenter creates an activity, the activity is added to the `bc_Workflow`, `bc_WorkflowLog` and `bc_WorkflowWorkItem` tables. Once a user completes the activity, BillingCenter sets the workflow status to completed. The `bc_Workflow`, `bc_WorkflowLog` and `bc_WorkflowWorkItem` table entry for the activity are never used again. These tables grow in size over time and can adversely affect performance as well as waste disk space. Excessive records in these tables also negatively impacts the performance of the database upgrade.

Remove workflows, workflow log entries, and workflow items for completed activities to improve database upgrade and operational performance and to recover disk space.

BillingCenter includes work queues to purge completed workflows and their logs that are older than a configurable number of days. Guidewire recommends that you purge completed workflows and their logs periodically. This reduces performance issues caused by having a large number of unused workflow log records.

To set the number of days after which the `purgeworkflows` process purges completed workflows and their logs, set the following parameter in `config.xml`:

```
<param name="WorkflowPurgeDaysOld" value="value" />
```

Set the value to an integer. By default, `WorkflowPurgeDaysOld` is set to 60. This is the number of days since the last update to the workflow, which is the completed date.

You can launch the Purge Workflows batch process from the `BillingCenter/admin/bin` directory with the following command:

```
maintenance_tools -password password -startprocess PurgeWorkflows
```

You can also purge only the logs associated with completed workflows older than a certain number of days. Run the `purgeworkflowlogs` process instead. This process leaves the workflow records and removes only the workflow log records. The `purgeworkflowlogs` process is configured using the `WorkflowLogPurgeDaysOld` parameter rather than `WorkflowPurgeDaysOld`.

You can launch the Purge Workflow Logs batch process from the `BillingCenter/admin/bin` directory with the following command:

```
maintenance_tools -password password -startprocess PurgeWorkflowLogs
```

Validating the Database Schema

This validation detects the unlikely event that the data model defined by your configuration files has become out of sync with the database schema. While the pre-upgrade server is running, use the `system_tools` command in `admin/bin` of the customer configuration to verify the database schema:

```
system_tools -password password -verifydbschema -server servername:port/instance
```

Correct any validation problems in the database before proceeding. Contact Guidewire Support for assistance.

Following the database upgrade, run this command again from the `admin/bin` directory of the target (upgraded) configuration.

Checking Database Consistency

BillingCenter has hundreds of internal database consistency checks. Before upgrading, run consistency checks to verify the integrity of your data.

Run database consistency checks early in the upgrade project. Fix any consistency errors. Continue to periodically run consistency checks and resolve issues so that your database is ready to upgrade when you begin the upgrade procedure. Consistency issues might take some time to resolve, so begin the process of running consistency checks and fixing issues early. Contact Guidewire Support for information on how to resolve any consistency issues.

After the database upgrade, run consistency checks again from the BillingCenter [Consistency Checks](#) page.

To run consistency checks

1. Start the BillingCenter server if it is not already running.
2. Log in to BillingCenter with an administrator account.
3. Press `Alt + Shift + T` to access the **Server Tools**.
4. Click **Info Pages**.
5. Select **Consistency Checks** from the drop-down list.
6. To increase the number of threads used to run consistency checks, increase the **Number of threads**. The number of threads to use depends on the capability of your database server. Increasing the number of threads can improve performance of consistency checks as long as your server can process the threads. Guidewire recommends starting with five threads. If too many threads are used, there is a greater chance that current users experience reduced performance if the database server is fully loaded.

To set the number of threads in versions prior to 8.0, specify a value for the `checker.threads` parameter within the `database` block of `config.xml`.

```
<database>
  ...
  <param name="checker.threads" value="5" />
  ...
</database>
```

7. Click **Run Consistency Checks**.

See also

“[Checking Database Consistency](#)” on page 36 in the *System Administration Guide*.

Creating a Data Distribution Report

Generate a data distribution report for the database before an upgrade. Save the output of this report. Run the report again after the upgrade to ensure the distribution is still correct.

Guidewire is very interested in the data distribution of your databases. Guidewire uses these reports to better understand the nature of your database and to optimize BillingCenter performance. Guidewire appreciate copies of your reports, both before and after upgrades.

You can also use this information to tune the application server cache. See “[Application Server Caching](#)” on page 63 in the *System Administration Guide*.

To create a database distribution report

1. In config.xml, set <param name="EnableInternalDebugTools" value="true"/>.
2. Start the BillingCenter application server.
3. Log into BillingCenter as an administrative user.
4. Type ALT + SHIFT + T while in any screen to reach the **Server Tools** page.
5. Choose **Info Pages** from the **Server Tools** tab.
6. Choose the **Data Distribution** page from the **Info Pages** dropdown.
7. Enter a reason for running the Data Distribution batch job in the **Description** field.
8. On this page, select the **Collect distributions for all tables** radio button and check all checkboxes to collect all distributions.
9. Push the **Submit Data Distribution Batch Job** button on this page to start the data collection.
10. Return to the **Data Distribution** page and push its **Refresh** button to see a list of all available reports. The batch job has completed when the **Available Data Distribution** list on the **Data Distribution** page includes your description.
11. Select the desired report and use the **Download** button to save it zipped to a text file. Unzip the file to view it.

Generating Database Statistics

To optimize the performance of the BillingCenter database, it is a good idea to update database statistics on a regular basis. Both SQL Server and Oracle can use these statistics to optimize database queries.

If you update database statistics on a regular basis, you do not need to update statistics before an upgrade. If you do not update database statistics on a regular basis, Guidewire recommends that you update incremental statistics before running the upgrade.

To generate incremental database statistics

1. Get the proper SQL statements for updating the statistics in BillingCenter tables by running the following command in the pre-upgrade environment:

```
maintenance_tools -getincrementaldbstatisticsstatements -password password  
-server http://server:port/instance > db_stats.sql
```
2. Run the resulting SQL statements against the BillingCenter database.

You can configure SQL Server to periodically update statistics. See your database documentation and “Configuring Database Statistics” on page 39 in the *System Administration Guide* for more information.

The database upgrade can take a long time, and has built-in statistics collection that help you see if any part of the upgrade is slow. Collect these statistics, and compare them to the statistics you collected before the upgrade. The config.xml file has parameters that control this statistics collection.

If you disabled statistics collection during the upgrade by setting updatestatistics to false, Guidewire recommends that you generate full statistics as soon as possible after the upgrade. For instructions, see “Commands for Updating Database Statistics” on page 40 in the *System Administration Guide*. Note that the commands for generating statistics have moved to system_tools instead of maintenance_tools in the upgraded BillingCenter.

Creating a Database Backup

Prepare the environment so that you can make a total recovery of the original installation if you run into problems during the upgrade.

The first time you start the BillingCenter server after running the upgrade tool, the server updates the database. During its work, the database upgrader minimizes the logging that it does. For these reasons, back up your database before starting an upgrade. Your pre-upgrade database might not be recoverable after an upgrade.

Updating Database Infrastructure

Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

Preparing the Database for Upgrade

This topic notes steps to prepare the database for the upgrade process.

Ensuring Adequate Free Space

The database upgrade requires significant free space. Make sure the database has at least 50% of the current database size available as free space.

Disabling Replication

Disable database replication during the database upgrade.

Assigning Default Tablespace (Oracle only)

Set the default tablespace for the database user to the one mapped to the logical tablespace OP in config.xml.

The database upgrade creates temporary tables during the upgrade without specifying the tablespace. If the Oracle database user was created without a default tablespace, Oracle by default creates the tables in the SYSTEM tablespace. The Guidewire database user is likely not to have the required quota permission on the SYSTEM tablespace. This results in an error of the type:

```
java.sql.SQLException: ORA-01950: no privileges on tablespace 'SYSTEM'
```

Even if the default tablespace is not SYSTEM, if the Guidewire database user does not have quota permission on the default tablespace, the temporary table creation during upgrade fails.

Setting Linguistic Search Collation

WARNING For SQL Server, compare the default collation of the database to the collation defined for your locale. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your SQL Server database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, this adds time to the total database upgrade process.

WARNING Oracle Java Virtual Machine (JVM) must be installed on all Oracle databases hosting BillingCenter. The only exception is when the BillingCenter application locale is English and you only require case-insensitive searches. Ensure that Oracle initialization parameter `java_pool_size` is set to a value of above 50 MB.

You can specify how you want BillingCenter to collate search results. The `strength` attribute of the `LinguisticSearchCollation` element of `GWLocale` for the default locale in `localization.xml` specifies how BillingCenter sorts search results. You can set the `strength` to `primary` or `secondary`.

With `LinguisticSearchCollation strength` set to `primary`, BillingCenter searches results in a case-insensitive and accent-insensitive manner. BillingCenter considers an accented character equal to the unaccented version of the character if the `LinguisticSearchStrength` for the default application locale is set to `primary`. For example, with `LinguisticSearchCollation strength` set to `primary`, BillingCenter treats “Renée”, “Renee”, “renee” and “reneé” the same.

With `LinguisticSearchCollation strength` set to `secondary`, BillingCenter searches results in a case-insensitive, accent-sensitive manner. BillingCenter does not consider an accented character equal to the unaccented version of the character if the `LinguisticSearchCollation strength` for the default application locale is set to `secondary`. For example, with `LinguisticSearchCollation strength` set to `secondary`, a BillingCenter search treats “Renee” and “renee” the same but treats “Renée” and “reneé” differently. By default, BillingCenter uses a `LinguisticSearchCollation strength` of `secondary`, for case-insensitive, accent-sensitive searching.

The `collations.xml` file defines the collations to use for different locales and different collation strengths. The `primary`, `secondary`, and `tertiary` attributes of the `Collation` element define the collation to use depending on the `LinguisticSearchCollation strength` attribute in `localization.xml`.

BillingCenter 7.0 introduced configurable linguistic searching for SQL Server databases. In releases prior to BillingCenter 7.0, BillingCenter used the collation setting of the database server. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, this adds time to the total database upgrade process.

For sorting search results, the following rules apply:

- **Case** – All searches ignore the case of the letters, whether `LinguisticSearchCollation strength` is set to `primary` or `secondary`. “McGrath” equals “mcgrath”.
- **Punctuation** – Punctuation is always respected, and never ignored. “O'Reilly” does not equal “OReilly”.
- **Spaces** – Spaces are respected. “Hui Ping” does not equal “HuiPing”.
- **Accents** – An accented character is considered equal to the unaccented version of the character if `LinguisticSearchCollation strength` is set to `primary`. An accented character is not equal to the unaccented version if `LinguisticSearchCollation strength` is set to `secondary`.

Japanese only

- **Half Width/Full Width** – Searches under a Japanese locale always ignore this difference.
- **Small/Large Kana** – Japanese small/large letter differences are ignored only when `LinguisticSearchCollation strength` is set to `primary`, meaning accent-insensitive.
- **Katakana/Hiragana sensitivity** – Searches under a Japanese locale always ignore this difference.
- The long dash character is always ignored.
- Soundmarks (` and °) are only ignored if `LinguisticSearchCollation strength` is set to `primary`.

German only

- Vowels with an umlaut compare equally to the same vowel followed by the letter e. Explicitly, “ä”, “ö”, “ü” are treated as equal to “ae”, “oe” and “ue”.
- The Eszett, or sharp-s, character “ß” is treated as equal to “ss”.

BillingCenter populates denormalized values of searchable columns to support the search collation. For example, with `LinguisticSearchCollation strength` set to `primary`, BillingCenter stores the value “Renée”, “Renée”, “renée” and “renée” in a denormalized column as “renée”. With `LinguisticSearchCollation strength` set to `secondary`, BillingCenter stores a denormalized value of “renée” for “Renée” or “renée” and stores “renée” for “Renée” or “renée”. Japanese and German locales make additional changes when storing values in denormalized columns in order to conform to the rules listed previously for those locales.

Any time you change the `LinguisticSearchCollation strength` and restart the server, BillingCenter repopulates the denormalized columns. Previous versions of BillingCenter populated the denormalized columns with lowercase values for case-insensitive search, equivalent to setting `LinguisticSearchCollation strength` to `secondary`. If you set `LinguisticSearchCollation strength` to `primary`, BillingCenter repopulates the denormalized columns, substituting any accented characters for their base equivalents. This process can take a long time, depending on the amount of data. Therefore, if you want to change `LinguisticSearchCollation strength` to `primary`, you might want to do so after the database upgrade. If you are concerned about the duration of the database upgrade, you can change your search collation settings after the upgrade. During a maintenance period, change `LinguisticSearchCollation strength` to `primary` and restart the server to repopulate the denormalized columns.

For Japanese locales, the BillingCenter database upgrade from a prior major version repopulates the denormalized columns regardless of the `LinguisticSearchCollation strength` value. BillingCenter must repopulate the denormalized columns for Japanese locales to have search results obey the Japanese-only rules listed previously.

Field Encryption and the Upgraded Database

The database upgrader handles most encrypted fields with no problem. If you have started to use field encryption and have changed fields in the database from no field encryption to encrypted, specifying some encryption algorithm, the upgrader preserves this encryption. However, if you later change the encryption to another algorithm type, the upgrader does not handle this case. See “[Changing Your Encryption Algorithm Later](#)” on page 258 in the *Integration Guide*.

Customizing the Upgrade

The `IDatamodelUpgrade` plugin interface provides hooks for custom code that you want to run during the database upgrade. You can use the `IDatamodelUpgrade` plugin to:

- execute custom version checks to test data or the data model itself before starting the upgrade.
- make custom database changes before or after the database upgrade.
- make data model changes to archived entities.

For example, you might fix a consistency check failure issue, correct issues reported by version checks, or delete a custom extension that you are no longer using.

IMPORTANT BillingCenter 3.0 included a similar plugin interface, `IDatabaseUpgrade`. If you previously implemented `IDatabaseUpgrade` for an upgrade to BillingCenter 3.0, you must now implement `IDatamodelUpgrade` if you want to execute custom upgrade code.

Running Custom Version Checks and Triggers

You can use the `IDatamodelUpgrade` plugin to run custom version checks and triggers before and after the database upgrade. The `IDatamodelUpgrade` plugin interface contains method signatures for two methods that you must define in your plugin. These signatures are:

- `property getBeforeUpgradeDatamodelChanges() : List<IDataModelChange<BeforeUpgradeVersionTrigger>>`
- `property getAfterUpgradeDatamodelChanges() : List<IDataModelChange<AfterUpgradeVersionTrigger>>`

Each method returns a list of `IDataModelChange` entities, each taking a `BeforeUpgradeVersionTrigger` or `AfterUpgradeVersionTrigger` type parameter. The `IDataModelChange` interface has two methods that you use to make data model changes. The `getDatabaseUpgradeVersionTrigger` method is for changes to the database. The `getArchivedDocumentUpgradeVersionTrigger` method is for changes to archived entities. If your organization has not implemented archiving or you do not want to make changes to archived entities, return null for `getArchivedDocumentUpgradeVersionTrigger`.

The `getAfterUpgradeDatamodelChanges` method runs after the Guidewire upgrade version triggers. You can use this method to move data into extension tables or columns that did not exist prior to upgrading.

You can return an empty list from either `getBeforeUpgradeDatamodelChanges` or `getAfterUpgradeDatamodelChanges`. For example, if you only have triggers to run before the upgrade, you can return an empty list from `getAfterUpgradeDatamodelChanges`.

Modifying Tables

Both `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` base classes provide a protected `getTable` method that accepts a `string` parameter. The `getTable` method returns an `IBeforeUpgradeTable` or `IAfterUpgradeTable` object that provides a number of methods for DDL and DML operations, such as:

- `create` – Create the table if it does not already exist. The table must be related to an entity defined in the data model. This method is available only for `IBeforeUpgradeTable`.
- `delete` - Deletes rows from a table. Returns a builder (`IBeforeUpgradeDeleteBuilder` for `IBeforeUpgradeTable`, `IDeleteBuilder` for `IAfterUpgradeTable`) that has methods for comparing data to restrict which rows are deleted.
- `drop` - Drops the table.
- `dropColumns` - Drops multiple columns from the table.
- `getColumn` – Returns an `IBeforeUpgradeColumn` or `IAfterUpgradeColumn` object that has methods to perform DDL operations on the column such as create, drop, rename, and more.
- `insert` – Returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable`) to perform an insert operation.
- `insertSelect` – Returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`) for SQL to perform an insert operation using data selected from a table.
- `rename` - Renames the table.
- `update` – Returns a builder (`IBeforeUpgradeUpdateBuilder` for `IBeforeUpgradeTable`, `IUpdateBuilder` for `IAfterUpgradeTable`) for SQL to perform an update operation.

For DML operations, call the `execute` method on the builder to actually perform the operation. The `execute` method runs in its own transaction. You do not need to handle transactions and `TransactionManager`.

There are more methods on the `IBeforeUpgradeTable` and `IAfterUpgradeTable` classes documented in the Guidewire Gosu API documentation. To generate the Guidewire Gosu API documentation, run the `gwbc regen-gosudoc` command from the BillingCenter `bin` directory. Then, open `BillingCenter/build/gosudoc/index.html`.

The methods for `BeforeUpgradeVersionTrigger` intentionally take strings but not entities or properties. This is because the name of the column could change in the future. Consider `PropertyA` on `EntityE` which corresponds to column A in the database. Suppose you use `PROPERTYA_PROP` in a version trigger at minor version 200, but at minor version 250, you decide to rename the backing column from A to B. The version trigger you wrote in the past would break because it would execute before the rename operation and would try to use the new column name.

`AfterUpgradeVersionTrigger` is very similar to `BeforeUpgradeVersionTrigger`. A few differences include:

- The `AfterUpgradeVersionTrigger` DML builders use the query builder, `IQueryBuilder`.
- In an `AfterUpgradeVersionTrigger` you can use properties and types in addition to strings.
- Some DDL operations are not provided on the `IAfterUpgradeTable` object, including creating a table or adding a column.

Unless you require one of the unique capabilities of `AfterUpgradeVersionTrigger`, use `BeforeUpgradeVersionTrigger` for custom version triggers.

Upgrading Typelists

The `BeforeUpgradeVersionTrigger` class includes a `getTypeKeyID` method with the following signature:

```
protected final Integer getTypeKeyID(IEntityType subtype)
```

Note: Protected methods do not appear in the Gosu documentation. Use CTRL + SPACE in Studio to show available methods and properties.

The `getTypeKeyID` method returns the integer ID of the type code in the type list matching the given table name. This method checks both the existing typelist tables and the metadata files to determine what all typekey IDs will be after upgrade. Therefore, the `getTypeKeyID` method works as expected even before a new typekey or typelist table is created during the automatic schema upgrade phase.

This method also works for orphaned typecodes that have not yet been removed from the database. These are typecodes that still exist in the database table but not in the metadata file. You can use the `getTypeKeyID` method for remapping usages of orphaned typecodes.

Version Checks

In some cases, you might want to check for a certain condition in the database before the upgrade proceeds. This is referred to as a version check. Only read operations are available in version checks. For example, you can write a version check to query a table or check the existence of a table or column, but the check cannot insert new rows. The `BeforeUpgradeVersionTrigger` class includes a `hasVersionCheck` method that you must define to return true or false. If the trigger does include a version check, overwrite the `createVersionCheck` method to define your custom version check. For standalone version checks that are not associated with a version trigger, you can use `BeforeUpgradeVersionCheckWrapper`.

The upgrade executes all custom version checks before custom version triggers. The upgrade runs Guidewire version checks after all custom `BeforeUpgradeVersionTrigger` implementations, so you can create a `BeforeUpgradeVersionTrigger` to correct issues detected by the Guidewire version checks.

If a custom version check fails, the upgrade stops before running any upgrade triggers. Correct the issue and restart the upgrade.

Order of Execution

The upgrade performs actions in the following order:

Step	Action	In the event of failure due to a data issue...
1	Custom version checks	Correct the data issue. Restart the upgrade. You do not need to restore the database because the upgrade has not made any changes.
2	Custom BeforeUpgradeVersionTrigger implementations	Restore the database from a backup. Correct the data issue. Consider adding custom version checks to test for other instances of the data issue.
3	Guidewire version checks	If you do not have any custom BeforeUpgradeVersionTrigger implementations, correct the data issue and restart the upgrade. If you do have custom BeforeUpgradeVersionTrigger implementations, restore the database from a backup. Then, correct the data issue. In either case, consider creating a custom BeforeUpgradeVersionTrigger implementation to correct the data issue.
4	Guidewire version triggers	A failure due to data issues at this stage is unlikely. Contact Guidewire Support.
5	Automated data model upgrade to update the database to the defined data model	A failure due to data issues at this stage is unlikely. Contact Guidewire Support.
6	Guidewire version triggers that require the updated data model in the database	A failure due to data issues at this stage is unlikely. Contact Guidewire Support.
7	Custom AfterUpgradeVersionTrigger implementations	Restore the database from a backup. Correct the data issue. Consider creating a custom BeforeUpgradeVersionTrigger implementation to correct the data issue if possible.

The preceding table describes failure cases that are caused by data issues. If the upgrade fails for other reasons, such as a disruption of the database server, fix the issue causing the disruption, restore the database, and restart the upgrade.

Versioning

Each `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` instance requires a minor version number, passed as an integer. If the data model version number is less than or equal to the number passed to the instance, then the trigger executes. Whenever you make a data model change, or you want to force an upgrade, increment the version number in `extensions.properties`.

To run custom version checks and triggers

1. Create a new package, such as `companyName.upgrade`, to store your custom version triggers.
 - a. Open Studio.
 - b. In the Studio Project window, expand configuration.
 - c. Right-click `gsrc` and click **New → Package**.
 - d. Enter a package name for upgrade purposes, such as `companyName.upgrade`.
2. Right-click the upgrade package and click **New → Gosu Class**.
3. Enter a name for the class and click **OK**.

- 4.** Create a new Gosu class that extends `CustomerDatamodelUpgrade` and implements `IDatamodelUpgrade`. The class you create must define the `getBeforeUpgradeDatamodelChanges` and `getAfterUpgradeDatamodelChanges` methods. This class is the container from which you call custom version trigger classes.

For example:

```
package companyName.upgrade
uses gw.plugin.upgrade.IDatamodelUpgrade
uses java.lang.Iterable
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger
uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger
uses java.util.ArrayList
uses gw.api.datamodel.upgrade.CustomerDatamodelUpgrade
uses gw.api.datamodel.upgrade.IDatamodelChange
uses gw.api.database.upgrade.DatamodelChangeWithoutArchivedDocumentChange

class TestDatamodelUpgradeImpl extends CustomerDatamodelUpgrade implements IDatamodelUpgrade {

    override property get BeforeUpgradeDatamodelChanges() :
        List<IDatamodelChange<BeforeUpgradeVersionTrigger>> {
            var list = new ArrayList<IDatamodelChange<BeforeUpgradeVersionTrigger>>()
            list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger1()))
            list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger2()))
            return list
        }

    override property get AfterUpgradeDatamodelChanges() :
        List<IDatamodelChange<AfterUpgradeVersionTrigger>> {
            var list = new ArrayList<IDatamodelChange<AfterUpgradeVersionTrigger>>()
            list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new AfterVersionTrigger1()))
            return list
        }
}
```

- 5.** Create your custom `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` Gosu classes. See “`IDatamodelUpgrade` API Examples” on page 146.

- 6.** Implement the `IDatamodelUpgrade` plugin with the new class.

- a. Start Guidewire Studio 8.0.4 by entering `gwbc studio` from the `BillingCenter/bin` directory.
- b. In Studio, expand `configuration` → `config` → `Plugins`.
- c. Right-click `registry` and click `New` → `Plugin`.
- d. In the `Plugin` dialog, enter the name `IDatamodelUpgrade`. For this plugin, the name must match the interface.
- e. In the `Plugin` dialog, click the ... button.
- f. In the `Select Plugin Class` dialog, type `IDatamodelUpgrade` and select the `IDatamodelUpgrade` interface.
- g. In the `Plugin` dialog, click `OK`. Studio creates a GWP file under `Plugins` → `registry` with the name you entered.
- h. Click the `Add Plugin` icon (a plus sign) and select `Add Gosu Plugin`.
- i. For `Gosu Class`, enter your class, including the package.
- j. Save your changes.

When you start the server to perform the database upgrade from a prior major version, the upgrade calls the plugin and runs your custom methods.

`IDatamodelUpgrade` API Examples

This topic first introduces the basic structure of a `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger`. Next it shows methods that can be included within the `execute` method of these triggers to modify the database.

This topic includes:

- “BeforeUpgradeVersionTrigger Structure” on page 147
- “AfterUpgradeVersionTrigger Structure” on page 148
- “Altering Columns to Match Data Model” on page 148
- “Altering a Non-nullable Column to Nullable” on page 148
- “Creating Columns” on page 149
- “Dropping Columns” on page 150
- “Renaming Columns” on page 150
- “Setting a Column Value for a Specific Subtype” on page 150
- “Creating Tables” on page 151
- “Renaming Tables” on page 151
- “Deleting Rows” on page 151
- “Inserting Rows” on page 152
- “Inserting Data Selected from Another Table” on page 152
- “Updating Rows” on page 153

BeforeUpgradeVersionTrigger Structure

A custom BeforeUpgradeVersionTrigger subclass has the following structure.

```
package companyName.upgrade.before

uses gw.api.database.upgrade.before.BeforeUpgradeVersionCheck
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger

class myBeforeUpgradeTrigger extends BeforeUpgradeVersionTrigger {

    construct() {
        super(dataModelVersionNumber)
    }

    override function execute() {
        // Perform actions here.
    }

    override function hasVersionCheck() : boolean {
        // return true if creating a version check to determine whether the trigger can run.
        // return false if you are not implementing a version check.
    }

    override property get Description() : String {
        return "Description of the version trigger."
    }

    // Override the createVersionCheck method if you are implementing a version check.
    override function createVersionCheck() : BeforeUpgradeVersionCheck {
        return new BeforeUpgradeVersionCheck(dataModelVersionNumber) {

            override function verifyUpgradability() {
                if (condition to detect) {
                    addVersionCheckProblem("description of issue")
                }
            }

            override property get Description() : String {
                return "Description of the version check."
            }
        }
    }
}
```

Define the execute method to perform the actions you want your custom trigger to perform. Some examples are provided in subsequent topics.

AfterUpgradeVersionTrigger Structure

A custom AfterUpgradeVersionTrigger subclass has the following structure.

```
package companyName.upgrade.after

uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger

class myAfterUpgradeTrigger extends AfterUpgradeVersionTrigger{

    construct() {
        super(dataModelVersionNumber)
    }

    override function execute() {
        // Perform actions here.
    }

    override property get Description(): String {
        return "Description of the version trigger."
    }
}
```

Altering Columns to Match Data Model

In most cases, you do not need to alter a column to match a change to the column type in the logical data model. The upgrader automatically applies data model changes to the database. However, this occurs after all custom BeforeUpgradeVersionTrigger instances have run, so Guidewire provides methods to alter database columns to match the data model.

Altering a single column

If you need to alter a single column for use in a BeforeUpgradeVersionTrigger, modify the data model file, then use the alterColumnTypeToMatchDatamodel method of IBeforeUpgradeColumn.

For example:

```
var table = getTable("TableName")
var column = table.getColumn("ColumnName")
column.alterColumnTypeToMatchDatamodel()
```

Altering multiple column

To alter multiple columns for use in a BeforeUpgradeVersionTrigger, modify the data model file, then use the alterMultipleColumnsToMatchDatamodel method of IBeforeUpgradeTable.

For example:

```
var table = getTable("TableName")
var columnsToChange = new IBeforeUpgradeColumn[2]

columnsToChange[0] = table.getColumn("column1")
columnsToChange[1] = table.getColumn("column1")

table.alterMultipleColumnsToMatchDatamodel(columnsToChange)
```

Altering a Non-nullable Column to Nullable

To alter a column from non-nullable to nullable, use the IBeforeUpgradeColumn method alterColumnTypeToNullable.

For example:

```
var table = getTable("TableName");
table.getColumn("ColumnName").alterColumnTypeToNullable();
```

Creating Columns

The database upgrader automatically creates a column that is added to the data model if the column meets one of the following criteria:

- Nullable
- Non-nullable with a default value specified in the metadata
- Non-nullable without a default value if there are no rows in the table
- The column is an editable field

However, you might want to explicitly create the column in your upgrade trigger if you want the trigger to perform an action on the column such as populating it.

In the data model, the column must be defined as a property on an entity. The database upgrade will determine the correct datatype and nullability from the data model.

Creating a new column is moderately expensive in terms of performance of the upgrade.

Creating a Column

To create a column, invoke the `create` method on the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")  
  
// Create column with given name.  
// Column must be backed by a property on an entity.  
// Upgrader will figure out the correct datatype and nullability.  
  
table.getColumn("ColumnName").create()
```

Creating a Non-nullable Column with an Initial Value

The upgrader throws an exception if you try to add a new non-nullable column without a default value and there are rows in the table. For non-nullable columns, either specify a default value, or create a version trigger that will populate the column.

To create a new column as non-nullable with an initial value, use the `createNonNullableWithInitialValue()` method. In the data model, the column must be defined as non-nullable.

For example:

```
IBeforeUpgradeTable table = getTable("TableName")  
table.getColumn("ColumnName").createNonNullableWithInitialValue(Initial value)
```

The initial value must be of the appropriate type for the column's datatype. You can alter this value in later steps as needed.

Creating a Temporary Column

Use the `createTempColumn` method of `IBeforeUpgradeTable` to add a temporary column to the table. The `createTempColumn` method takes two parameters, a `String` for the column name and an `IDataType` for the column data type. `createTempColumn` creates a new nullable column with the given name and datatype to hold temporary data. You must explicitly drop the temporary column during the upgrade. The schema verifier will report an error during server startup if the column has not been dropped. You can create the temporary column in a `BeforeUpgradeVersionTrigger` and drop it in an `AfterUpgradeVersionTrigger`. This approach is useful when you want to move data from a column that will be removed during the upgrade to a column that will be created during the upgrade.

In the following example, a `BeforeUpgradeVersionTrigger` adds a temporary `shorttext` column to an existing entity and populates it with data from another column on a different entity. An `AfterUpgradeVersionTrigger` moves the data to a new entity.

BeforeUpgradeVersionTrigger Execute Method

```
// Add a temporary column to TableA.
var tableA = getTable("TableA")
var tempColumn = tableA.createTempColumn("tmp_column", DataTypes.shorttext())

// Get an IBeforeUpgradeUpdateBuilder for TableA.
var ub = tableA.update()

// Set the value of the temporary column to the value of ColumnA.
ub.set(tempColumn, ub.getColumnRef("ColumnA"))

ub.execute()
```

AfterUpgradeVersionTrigger Execute Method

```
// Get an IUpdateBuilder for TableA.
var ub = getTable("TableA").update().withLogSQL(true)

var q = new Query(Account).withLogSQL(true)
q.compare("ID", Equals, ub.getQuery().getColumnRef("Account"))
var piDesc = PaymentInstrument.Type.TypeInfo.getProperty("Description") as IEntityPropertyInfo

ub.set(piDesc, q, q.getColumnRef(DBFunction.Expr({"tmp_xyz"}))) // tmp_xyz is the DB table column name
ub.execute()

var tempColumn = getTable("someTable").getColumn("tmp_xyz").drop()
```

Dropping Columns

The upgrader does not drop existing columns in order to prevent data loss. You can write a version trigger to move the data (not shown in example) and then drop the column by using the `drop()` method of the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")
table.getColumn("ColumnName").drop()
```

There is a `dropColumns` method on `IBeforeUpgradeTable` to drop multiple columns in one statement. The `dropColumns` method takes an array of `IBeforeUpgradeColumn` objects.

For example:

```
var table = getTable("TableName")
table.dropColumns(table.getColumn("ColumnName2"), table.getColumn("ColumnName3"));
```

In Oracle, dropping a column usually has little effect on upgrade performance. Dropping a column actually marks the column as unused in the metadata. At a later point, the DBA is responsible for performing the necessary cleanup. You can override this functionality and force columns to be dropped right away.

In SQL Server, dropping a column is performance-intensive because the RDBMS has to do some clean up work.

Renaming Columns

To rename a column use the `rename` function on the column object.

```
override function execute() {
    getTable("TableName").getColumn("ColumnName").rename("NewColumnName")
}
```

Setting a Column Value for a Specific Subtype

To set a column to a specific value for specific subtypes, use the `set` and `compare` methods of an `IBeforeUpgradeTable`. Get the typekey ID for comparison using the `BeforeUpgradeVersionTrigger` method `getTypekeyID`.

```
final var myTable = getTable("tableName")
final var myTypecode = getTypeKeyID("typelist name", "typelist code")

final var updateBuilder = myTable.update()

updateBuilder
    .set("myColumn", "some value")
    .compare("subtype", Equals, myTypecode)
```

```
updateBuilder.execute()
```

Creating Tables

To add a new table to the database, define a new entity in the data model. The upgrade creates the table automatically. However, you might want to explicitly create the table in your upgrade trigger if you want the trigger to perform an action on the table such as populating it.

Creating a new table has negligible impact on upgrade performance.

You can create a regular table using the `create` method of `IBeforeUpgradeTable`. The table must first be defined in the data model.

For example:

```
var table = getTable("TableName").create()
```

Creating Temporary Tables

You can add a temporary table to the database based on either the current database schema for a table or the data model definition of a table. You can also create a temporary table with a custom definition.

To create a temporary table based on the current table schema in the database, use the `createNewTempTableBasedOnCurrentSchema` method of `IBeforeUpgradeTable`. The table must be associated with an entity and exist in the database. The returned temporary table will contain the columns that this table has in the database currently. The columns may not match those specified in the entity metadata. For example, the metadata might contain a new column that has not yet been created. The `createNewTempTableBasedOnCurrentSchema` method is usually more appropriate than `createNewTempTableBasedOnThis` if you want to copy data from this table into the new temporary table as the columns will match exactly.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnCurrentSchema()
```

To create a temporary table based on the entity definition of a table in the data model, use the `createNewTempTableBasedOnThis` method of `IBeforeUpgradeTable`. Columns that do not exist in the table are not created on the temporary table, even if the metadata defines such a column. This table may not contain columns that are going to be renamed. The metadata reflects the new name for the column but does not have an entry for the old name, so it would not be added to the temporary table.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnThis()
```

To create a temporary table with a custom definition, use the `createAsNewTempTable` method of `IBeforeUpgradeTable`. This method takes a `Pair` array in which the first object is a `String` defining the column name and the second object is an `IDataType` defining the column data type.

Renaming Tables

To rename a table use the `rename` function on the table object.

```
override function execute() {
    getTable("extTableName").rename("TableName_EXT")
}
```

Deleting Rows

To delete rows from a table, use the `delete` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `delete` method returns a delete builder (`IBeforeUpgradeDeleteBuilder`) that provides methods for comparing column data to restrict the rows that are deleted.

In the following example, all rows that have a columnA value of 0 are deleted.

```
var table = getTable("SomeTable")
var deleteBuilder = table.delete()
deleteBuilder.Query.compare("columnA", Equals, 0)
deleteBuilder.execute()
```

Inserting Rows

To insert rows of data use the `insert` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insert` method returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable`) for SQL to perform an insert operation.

In the following example, an `IBeforeUpgradeInsertBuilder` is used to add two rows with three columns to table `myTable`. The `IBeforeUpgradeInsertBuilder` includes a description.

```
var myTable = getTable("SomeTable")
var insertBuilder = myTable.insert().withDescription("A custom insert
trigger to add two rows.")

insertBuilder
.mapColumn("columnA", "value of column A for first row")
.mapColumn("columnB", "value of column B for first row")
.mapColumn("columnC", "value of column C for first row")

insertBuilder.execute()

// add a second row
insertBuilder
.mapColumn("columnA", "value of column A for second row")
.mapColumn("columnB", "value of column B for second row")
.mapColumn("columnC", "value of column C for second row")

insertBuilder.execute()
```

Inserting Data Selected from Another Table

To insert data selected from another table use the `insertSelect` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insertSelect` method returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`). The builder includes a `mapColumn` method that can be passed explicit values, columns, or a query.

In the following example, the trigger sets `targetTable.column1` to an explicit value. The trigger sets `targetTable.column2` to the value of `sourceTable.sourceColumn`. Because there is no comparison being performed, the trigger will insert a row in the target table for each row in the source table:

```
var sourceTable = getTable("sourceTable")
var targetTable = getTable("targetTable")

var insertSelectBuilder = targetTable.insertSelect(sourceTable)

insertSelectBuilder.mapColumn("column1", "value") // sets a hard-coded value
.mapColumn("column2", sourceTable.getColumn("sourceColumn")) // sets column2 on target table to
// source table sourceColumn

insertSelectBuilder.execute()
```

In the next example, an existing table, `sourceTable`, is split into two tables, `targetTable1` and `targetTable2`.

```
var sourceTable = getTable("sourceTable")
var targetTable1 = getTable("targetTable1")
var targetTable2 = getTable("targetTable2")

var insertSelectBuilder1 = targetTable1.insertSelect(sourceTable)
var insertSelectBuilder2 = targetTable2.insertSelect(sourceTable)

insertSelectBuilder1.mapColumn("column1", sourceTable.getColumn("sourceColumn1"))
.mapColumn("column2", sourceTable.getColumn("sourceColumn2"))

insertSelectBuilder1.execute()
```

```
insertSelectBuilder2.mapColumn("column1", sourceTable.getColumn("sourceColumn3"))
    .mapColumn("column2", sourceTable.getColumn("sourceColumn4"))

insertSelectBuilder2.execute()
```

Updating Rows

To update rows in a table, use the update method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. This method returns a builder (`IBeforeUpgradeUpdateBuilder` or `IUpdateBuilder`). The builder includes methods to compare data to restrict which rows are updated.

In the following example, table `SomeTable` is updated to set `column1` to `SomeValue` for each row where the subtype matches a certain entity type:

```
var table = getTable("SomeTable")

// get IBeforeUpgradeUpdateBuilder
var ub = table.update()

// set column 1 to SomeValue
ub.set("column1", "SomeValue")
// where
    .compare("subType", Equals, getTypeKeyID(EntityType))
    .execute()
```

Running the Commission Payable Calculations Process

Run the Commission Payable Calculations process on your starting version before you upgrade the database. The Commission Payable Calculations process makes commission payable on direct bill policies.

To run the Commission Payable Calculations process

1. Start your pre-upgrade BillingCenter server.
2. Open BillingCenter and log in with an administrator account.
3. Open Server Tools by pressing ALT + SHIFT + T.
4. Click Batch Process Info.
5. In the Action column for Commission Payable Calculations, click Run. Wait for the process to complete before upgrading BillingCenter.

For more information about the Commission Payable Calculations process, see “Commission Payable Calculations Batch Processing” on page 126 in the *System Administration Guide*.

Configuring the Database Upgrade

You can set parameters for the database upgrade in the `BillingCenter 8.0.4 database-config.xml` file. The `<database>` block in `database-config.xml` contains parameters for database configuration, such as connection information. The `<database>` block contains an `<upgrade>` block that contains configuration information for the overall database upgrade. The `<upgrade>` block also contains a `<versiontriggers>` element for configuring general version trigger behavior and can contain `<versiontrigger>` elements to configure each version trigger.

This topic describes the parameters you can set for the database upgrade. For general database connection parameters, see “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.

Adjusting Commit Size for Encryption

You can adjust the commit size for rows requiring encryption by setting the `encryptioncommitsize` attribute to an integer in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade encryptioncommitsize="10000">
    ...
  </upgrade>
</database>
```

If BillingCenter encryption is applied on one or more attributes, the BillingCenter database upgrade commits batches of encrypted values. The upgrade commits `encryptioncommitsize` rows at a time in each batch. The default value of `encryptioncommitsize` varies based on the database type. For Oracle, the default is 10000. For SQL Server, the default is 100.

Test the upgrade on a copy of your production database before attempting to upgrade the actual production database. If the encryption process is slow, and you cannot attribute the slowness to SQL statements in the database, try adjusting the `encryptioncommitsize` attribute. After you have optimized performance of the encryption process, use that `encryptioncommitsize` when you upgrade your production database.

Configuring Version Trigger Elements

The database upgrade executes a series of version triggers that make changes to the database to upgrade between versions. You can set some configuration options for version triggers in `database-config.xml`. Normally, the default settings are sufficient. Change these settings only while investigating a slow database upgrade.

The `<database>` element in `database-config.xml` contains an `<upgrade>` element to organize parameters related to database upgrades. Included in the `<upgrade>` element is a `<versiontriggers>` element, as shown below:

```
<database ...>
  <param ... />
  <upgrade>
    <versiontriggers dbmsperfinfotreshold="600" />
  </upgrade>
</database>
```

The `<versiontriggers>` element configures the instrumentation of version triggers. This element has one attribute: `dbmsperfinfotreshold`. The `dbmsperfinfotreshold` attribute specifies for all version triggers the threshold after which the database upgrader gathers performance information from the database. You specify `dbmsperfinfotreshold` in seconds, with a default of 600. If a version trigger takes longer than `dbmsperfinfotreshold` to execute, BillingCenter:

- queries the underlying database management system (DBMS).
- builds a set of html pages with performance information for the interval in which the version trigger was executing.
- includes those html pages in the upgrader instrumentation for the version trigger.

You can completely turn off the collection of database snapshot instrumentation for version triggers by setting the `dbmsperfinfotreshold` to 0 in `config.xml`.

The `<versiontriggers>` element can contain optional `<versiontrigger>` elements for each version trigger. Each `<versiontrigger>` element can contain the following attributes.

Attribute	Type	Description
<code>name</code>	String	The case-insensitive name of a version trigger.
<code>extendedquerytracingenabled</code>	Boolean	Oracle only. Controls whether or not to enable extended sql tracing (Oracle event 10046) for the SQL statements that are executed by the version trigger. Default is <code>false</code> . The output can be very useful when debugging certain types of performance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation.
<code>parallel-dml</code>	Boolean	Oracle only. See “Configuring Parallel DML and DDL Statement Execution” on page 156.
<code>queryoptimizertracingenabled</code>	Boolean	Oracle only. Controls whether or not to enable query optimizer tracing (Oracle event 10053) for the SQL statements that are executed by the version trigger. Default is <code>false</code> . The output can be very useful when debugging certain types of performance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation.
<code>recordcounters</code>	Boolean	Controls whether the DBMS-specific counters are retrieved at the beginning and end of the use of the version trigger. Default is <code>false</code> . If true, then BillingCenter retrieves the current state of the counters from the underlying DBMS at the beginning of execution of the version trigger. If the execution of the version trigger exceeds the <code>dbmsperfinfotreshold</code> , then BillingCenter retrieves the current state of the counters at the end of the execution of the version trigger. BillingCenter writes differences to the DBMS-specific instrumentation pages of the upgrade instrumentation.
<code>updatejoinorderedhint</code>	Boolean	Oracle only. Whether to use the ORDERED hint for the UPDATE of a join. Default is <code>false</code> .
<code>updatejoinusemergehint</code>	Boolean	Oracle only. Whether to use the USE_MERGE hint for the UPDATE of a join. Default is <code>false</code> .
<code>updatejoinusenlhint</code>	Boolean	Oracle only. Whether to use the USE_NL hint for the UPDATE of a join. Default is <code>false</code> .

Deferring Creation of Nonessential Indexes

You can configure the upgrade to defer creation of nonessential indexes during the upgrade process until the upgrade completes and the application server is online. Nonessential indexes are performance-related indexes that do not enforce constraints. Creation of nonessential indexes can add significant time to the upgrade duration, so it is possible to defer this process. By default, the upgrade does not defer creation of these indexes.

To configure the upgrade to defer creation of nonessential indexes set the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` to `true`.

```
<database ...>
  <upgrade defer-create-nonessential-indexes="true">
    ...
  </upgrade>
</database>
```

If you opt to defer creation of nonessential indexes, BillingCenter runs the `DeferredUpgradeTasks` batch process as soon as the upgrade completes and the server is completely started. The `DeferredUpgradeTasks` batch process creates the nonessential performance indexes. The database user must have permission to create indexes until after the `DeferredUpgradeTasks` batch process is complete.

Deferring nonessential index creation can shorten the duration of the upgrade process. The BillingCenter database is then available sooner for tasks including upgrade verification and backing up the upgraded database before the database is opened up for production use. To take advantage of this earlier availability, perform upgrade testing and validation tasks while the `DeferredUpgradeTasks` batch process is running. Do not go into full production while the process is still running. The lack of so many performance-related indexes could likely make the system unusable.

Until the `DeferredUpgradeTasks` batch process has run to completion, BillingCenter reports errors during schema validation when starting. These include errors for column-based indexes existing in the data model but not in the physical database and mismatches between the data model and system tables.

Check the status of the `DeferredUpgradeTasks` batch process to determine when it has completed successfully. You can find the status of the deferred upgrade in the upgrade logs and on the BillingCenter [Upgrade Info](#) page. If the `DeferredUpgradeTasks` batch process fails, manually run the batch process again during non-peak hours.

If you do not opt to defer creation of nonessential indexes, BillingCenter creates these indexes as part of the upgrade process that must complete before the application server is online. If you do not want to defer creating nonessential indexes, the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` must be set to `false`. This is the default setting.

Configuring the Upgrade on Oracle

Configuring Column Removal

The database upgrade removes some columns. For Oracle, you can configure whether the removed columns are dropped immediately or are marked as unused. Marking a column as unused is a faster operation than dropping the column immediately. However, because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments. You can drop the unused columns after the upgrade during off-peak hours to free the space. Or, you can configure the database upgrade to drop the columns immediately during the upgrade. By default, the BillingCenter database upgrade marks columns as unused.

To configure the BillingCenter upgrade to drop columns immediately during the upgrade, set the `deferDropColumns` attribute of the `<upgrade>` block in `database-config.xml` to `false`. For example:

```
<database ...>
  ...
  <upgrade deferDropColumns="false">
    ...
  </upgrade>
</database>
```

By default, `deferDropColumns` is `true`.

Configuring Parallel DML and DDL Statement Execution

You can configure whether the upgrade executes DML (Data Manipulation Language) and DDL (Data Definition Language) statements in parallel or not and the degree of parallelism to use.

The `<upgrade>` element includes an `ora-parallel-dml` attribute. This attribute can be set to `disable`, `enable`, or `enable-all`. The default value is `enable`. If `ora-parallel-dml` is set to `disable`, the upgrade does not conduct parallel execution of DML statements. If `ora-parallel-dml` is set to `enable`, the upgrade executes DML statements in parallel if configured or coded for a version trigger. If `ora-parallel-dml` is set to `enable-all`, the upgrade executes DML statements in parallel in all cases unless turned off in the code or configuration for a version trigger.

The Boolean attribute `parallel-dml` of a `<versiontrigger>` element controls parallel execution for that version trigger. If `parallel-dml` is not set, the upgrade executes parallel DML statements if coded or if `ora-parallel-dml` is set to `enable_all` on the `<upgrade>` element. If `parallel-dml` is set to `false`, the upgrade does not execute DML statements in parallel. If `parallel-dml` is set to `true`, the upgrade executes DML statements in parallel if `ora-parallel` is set to `enable` or `enable_all`.

To configure the degree of parallelism for insert, update and delete operations, set the `degree-of-parallelism` attribute on the `<upgrade>` element. To configure the degree of parallelism for commands such as creating an index and enabling constraints using the alter table command, set the `degree-parallel-ddl` attribute on the `<upgrade>` element.

You can specify a value from 2 to 1000 to force that degree of parallelism. Specify a value of 1 to disable the use of parallel execution.

Setting either parameter to 0 configures BillingCenter to defer to Oracle to determine the degree of parallelism for the operations that attribute configures. The Oracle automatic parallel tuning feature determines the degree based on the number of CPUs and the value set for the Oracle parameter `PARALLEL_THREADS_PER_CPU`.

The default for both attributes is 4.

You can configure parallel DML execution on the `InsertSelectBuilder`, `BeforeUpgradeUpdateBuilder` and `BeforeUpgradeInsertSelectBuilder` of a custom version trigger using the `withParallelDml(boolean)` method. If not explicitly set to `true` or `false`, the upgrade uses parallel execution if configured. If set to `false`, the upgrade does not use parallel execution unless set to `true` for that version trigger. If set to `true`, it will be done unless set to `false` for that version trigger or `ora-parallel-dml` is set to `disable`.

Collecting Tablespace Usage and Object Size

To enable collection of tablespace usage and object size data on Oracle, set the `collectstorageinstrumentation` attribute of the `<upgrade>` block to `true`. For example:

```
<database ...>
...
<upgrade collectstorageinstrumentation="true">
...
</upgrade>
</database>
```

A value of `true` enables BillingCenter to collect tablespace usage and size of segments such as tables, indexes and LOBs (large object binaries) before and after the upgrade. The values can then be compared to find the utilization change caused by the upgrade.

Disabling Oracle Logging

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
...
<upgrade allowUnloggedOperations="true">
...
</upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run statements with the `NOLOGGING` option.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. Some examples include Reporting, Disaster Recovery through Standby databases and Oracle Dataguard. To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

Disabling Statistics Update for the Database

Generating table statistics during upgrade is optional for Oracle databases. The overall time required to upgrade the database is shorter if the database upgrade does not update statistics. To disable statistics generation during the upgrade, set the `updatestatistics` attribute of the `<upgrade>` element to `false`:

```
<upgrade updatestatistics="false">
```

If `updatestatistics` is `true`, the upgrade updates statistics and deletes histograms on columns for which BillingCenter does not generate statistics. This setting enables the upgrade to update statistics on changed objects. It also configures BillingCenter to maintain column-level statistics consistent with what is allowed in the code, data model and configuration.

If statistics are not updated during the upgrade, BillingCenter reports a warning that recommends that you run the database statistics batch process in incremental mode. Additionally, the [Upgrade Info](#) page shows that statistics were not updated as part of the upgrade. If statistics generation was not disabled, the [Upgrade Info](#) page reports the runs of the statistics batch process, including incremental runs.

You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded BillingCenter in a production environment. If you defer generating statistics during the upgrade, Guidewire recommends that you generate full statistics as soon as possible after the upgrade. For instructions, see “[Commands for Updating Database Statistics](#)” on page 40 in the *System Administration Guide*.

The [Upgrade Info](#) page does not identify the following case: You ran an upgrade with `updatestatistics=true` after running a previous upgrade with `updatestatistics=false`, but you did not update statistics first.

When you click the **Download** button on the [Upgrade Info](#) page, you get a more detailed report. This report shows the value of the `updatestatistics` attribute at the time of upgrade. Additionally, the report shows the update statistics SQL statements that were skipped as part of the upgrade. These statements are provided for reference. You typically do not need to review these statements if you run the incremental database statistics process following the upgrade.

Disabling Statistics Update for Tables with Locked Statistics

If you have tables that have locked statistics, specify to keep statistics on these tables before starting the database upgrade. To specify to keep statistics on a table, set the `action` attribute of the `<tablestatistics>` element for that table to `keep`. The `<tablestatistics>` element is nested within the `<databasestatistics>` element, which is within the `<database>` element in `database-config.xml`.

For example, if statistics are locked on `bc_someTable_EXT`, specify a `<tablestatistics>` element for that table with the `action` attribute set to `keep`:

```
<database>
  ...
  <databasestatistics>
    <tablestatistics name="bc_someTable_EXT" action="keep" />
  </databasestatistics>
</database>
```

Configuring the Upgrade on SQL Server

Disabling SQL Server Logging

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="true">
    ...
  </upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run with minimal logging. This can improve the performance of the upgrade. During the upgrade, set the SQL Server recovery model to Simple or Bulk logged. Once the upgrade and deferred upgrade tasks are complete, you can revert the recovery model setting and back up the full database.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. If you require full logging due to the presence of solutions such as Database Mirroring, continue to use the Full recovery model and set `allowUnloggedOperations` to `false`.

To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

Storing Temporary Sort Results in tempdb

For SQL Server databases, you can specify to store temporary sort results in tempdb by setting the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` block to `true`. By using tempdb for sort runs, disk input and output is typically faster, and the created indexes tend to be more contiguous. By default, `sqlserverCreateIndexSortInTempDB` is `false` and sort runs are stored in the destination filegroup.

If you set `sqlserverCreateIndexSortInTempDB` to `true`, you must have enough disk space available to tempdb for the sort runs, which for the clustered index include the data pages. You must also have sufficient free space in the destination filegroup to store the final index structure, because the new index is created before the old index is deleted. Refer to <http://msdn.microsoft.com/en-us/library/ms188281.aspx> for details on the requirements to use tempdb for sort results.

Specifying Filegroup to Store Sort Results for Clustered Indexes

For SQL Server databases, a version trigger recreates non-clustered backing indexes for primary keys as clustered indexes.

Before recreating the indexes, the version trigger automatically drops (and later rebuilds) any referencing foreign keys and drops any clustered indexes on tables with a primary key.

If you are using filegroups, the upgrade recreates the clustered index in the OP filegroup. By default, the upgrade also stores the intermediate sort results that are used to build the index in the OP filegroup. You can configure the upgrade to instead use the tempdb filegroup for the intermediate sort results.

If you want the upgrade to stores the intermediate sort results in the tempdb filegroup, set the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` element to `true`.

```
<database ...>
  ...
  <upgrade sqlserverCreateIndexSortInTempDB="true" />
  ...
</upgrade>
</database>
```

This option increases the amount of temporary disk space that is used to create an index. However, it might reduce the time that is required to create or rebuild an index when tempdb is on a different set of disks from that of the user database.

By default, `sqlserverCreateIndexSortInTempDB` is `false`.

Downloading Database Upgrade Instrumentation Details

The database upgrade deletes upgrade instrumentation information for prior database upgrades. If the database upgrade detects any prior upgrade instrumentation data, it reports a warning and deletes the data. If you have run previous database upgrades, and you want to preserve upgrade instrumentation details, follow the procedure in “Viewing Detailed Database Upgrade Information” on page 182.

Checking the Database Before Upgrade

The upgrade runs a series of version checks prior to making any changes to the database. These version checks ensure that the database is in a state that can be upgraded. Guidewire includes a number of version checks with BillingCenter and you can also add custom version checks.

You can configure BillingCenter to run the version checks only, including custom version checks. Before upgrading the production database, run version checks on a clone of your production database to identify any issues with your data.

To run version checks without database upgrade

1. Start Studio for BillingCenter 8.0.4 by running the following command from the bin directory:
`gwbc studio`
2. Expand **configuration** → **config** and open **database-config.xml**.
3. Add the attribute `versionchecksonly=true` to the `database` element. The `versionchecksonly` attribute overrides the `autoupgrade` attribute. If both are set to true, BillingCenter only runs version checks when the server starts.
4. Verify that the database connection is pointing to a clone of your production database.
5. Save your changes.
6. Start the server.

BillingCenter reports the number of version check errors. For any errors reported BillingCenter reports which version check resulted in the error along with the error message.

If BillingCenter reports version check errors, fix the data and rerun the version checks. Repeat this process until no errors are reported on the production clone. Apply the fixes to your production database prior to upgrade.

With `versionchecksonly=true` set, BillingCenter runs all version checks regardless of a failure in one of the checks. During a regular upgrade, BillingCenter stops the upgrade if an error is detected.

After you have fixed all version check errors, set `versionchecksonly` to `false` to run the actual upgrade.

Disabling the Scheduler

Before you start the server to upgrade the database, disable the scheduler for batch processes and work queues. Disabling the scheduler prevents batch processes and work queues from launching immediately after the database upgrade.

To disable the scheduler

1. Open the BillingCenter 8.0.4 `config.xml` file in a text editor.
2. Set the `SchedulerEnabled` parameter to `false`.
`<param name="SchedulerEnabled" value="false"/>`
3. Save `config.xml`.

After you have successfully upgraded the database, you can enable the scheduler by setting `SchedulerEnabled` to `true`. This can be accomplished by performing the database upgrade using a WAR or EAR file that has the `SchedulerEnabled` parameter to `false`. After the upgrade is complete and verified, stop the server and deploy a new WAR or EAR file that differs from the first only by having `SchedulerEnabled` set to `true`. Finally, restart the server to activate the scheduler.

Suspending Message Destinations

Suspend all event message destinations before you upgrade the database to prevent BillingCenter from sending messages until you have verified a successful database upgrade.

To suspend message destinations

1. Start the BillingCenter server for the pre-upgrade version.
2. Log in to BillingCenter with an account that has administrative privileges, such as the superuser account.
3. Click the **Administration** tab.
4. Click **Event Messages**.
5. Select the check box to the left of the **Destination** column to select all message destinations.
6. Click **Suspend**.

Resume messaging after you have verified a successful database upgrade.

Starting the Server to Begin Automatic Database Upgrade

The database upgrade is an automatic process that occurs as you start the server with the upgraded configuration of a new BillingCenter version. The database upgrade normally completes in a few hours or less.

If the database upgrade stops before completing, then restore your database from the backup, correct any issues reported, and repeat the database upgrade.

IMPORTANT Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

WARNING Except for your first database upgrade trials, do not start the server until you have upgraded all rules. Otherwise, default validation rules execute. This could strand objects at a high validation level and make it impossible to edit parts of the object.

WARNING The database upgrade runs a series of version checks prior to making any changes. If any of these checks fail, the upgrade aborts and reports an error message. You can fix the issue, create an updated backup of the database and attempt the upgrade again without restoring from a backup. However, if you experience a failure during the version triggers or upgrade steps portion of the upgrade, refresh the database from a backup before attempting the upgrade again.

Before you start the server to begin the database upgrade, follow the procedure in “Updating Rounding Mode Parameter” on page 125.

Test the Database Upgrade

Prior to attempting the database upgrade on a full-production database clone, test the database upgrade.

To test the database upgrade

1. Connected to the built-in Quickstart database, successfully start the built-in Quickstart application server with a merged configuration data model, including merged extensions, data types, field validators, and so forth.
2. Connected to an empty database on an Oracle or SQL Server database server, successfully start the Quickstart application server from the preceding step.
3. Connected to a restored backup of a production clone, start either the same Quickstart server from the preceding step or a supported third-party application server with your custom configuration.

In a development environment the database upgrade process records checkpoints of upgrade triggers that complete successfully. You can restart a failed database upgrade, and it resumes with the upgrade trigger that failed. This restart feature helps you test the upgrade with realistically large data sets. You avoid time spent to restore the database and rerun upgrade triggers that worked successfully.

Guidewire provides this feature for convenience while testing. However, it does not work for all failure scenarios. Even in development mode, under certain scenarios, you will have to restore a backup of the database taken prior to the upgrade attempts and then run the upgrade.

The database upgrade writes SQL executed by the failed trigger to the console. To restart a test database upgrade from a checkpoint reached in an earlier upgrade, manually roll back any database changes that occurred during the upgrade trigger that failed. Resolve the problem that caused the trigger to fail. Then start the server again to restart the upgrade. The upgrade skips successful upgrade triggers and continues by rerunning the trigger that failed.

A test run of your upgrade is successful only when it runs from start to finish without a restart.

WARNING Never use the restart feature of database upgrade in a production environment.

Integrations and Starting the Server

Disable all integrations during the automatic database upgrade. Integration points might require updates due to changes in Guidewire APIs. See the *BillingCenter New and Changed Guide* for specifics.

It is not necessary to have completely migrated integrations before attempting to start the server for the first time. If you have integrations that rely on non-Guidewire applications, do not expect these integrations to work the first time you start the server.

Understanding the Automatic Database Upgrade

As the database upgrade proceeds, it logs messages to the console as well as the log file describing its progress. The database upgrade process requires thousands of steps, divided into three phases. Due to the relational nature of a database, these phases must execute in a specific order for the upgrade to succeed.

During the first phase, the upgrader first executes custom `BeforeUpgradeVersionTrigger` version checks and triggers defined in the `IDataModelUpgrade` plugin. The upgrader next runs version checks defined by Guidewire. Then, the upgrader uses a set of version triggers defined by Guidewire to determine the actions that are required. The database upgrader requires version triggers in order to perform the following types of tasks:

- changing a datatype (other than just length)
- migrating data
- dropping a column
- dropping a table
- renaming a column
- renaming a table

Specific version triggers are described in this topic.

Many version triggers have version checks associated with them. These checks ensure that the database is ready for the associated version trigger. The database upgrade runs all checks before running any version triggers. If a check detects a problem, it reports the issue, including a sample SQL query to find specific problematic records. If a version check discovers an issue, the database upgrade stops before any version triggers are run. Therefore, it is not necessary to restore the database from a backup if a version check reports an error. Correct the issue and then create a new backup of the database. Then, if you encounter errors after the version check stage, you can restore a version of your database with the issue reported by the version check resolved.

In the second phase, the upgrader compares the target data model and the current database to determine how they differ. The upgrader makes changes to the database that do not require a version trigger during this phase.

Following this process, the third phase runs a subsequent set of version triggers. These triggers create actions that must be run last due to a dependency on an earlier phase.

After the database upgrade concludes, it reports issues that the upgrader encountered and did not complete.

You are responsible for correcting these issues. This might involve modifying the data model or altering the table manually. If you do not correct them, the next time you start the server you do *not* see a message that the database and the data model are out of sync. You must then use the `system_tools` command to verify the database schema.

Note: Given the complexity of database upgrade, Guidewire does not expose specific upgrade actions/steps to clients either in SQL or Java form. Any manual attempts to recreate or control the upgrade process can result in problems in the BillingCenter database. Recovery from such attempts is not supported.

Version Trigger Descriptions

The database upgrade uses version triggers to perform the actions described by sections within this topic. If a version trigger has an associated version check, the check is described with the trigger. Review these descriptions to familiarize yourself with some of the changes and to understand version checks. If a version check reports an issue, review the error message and consult the description of the relevant version trigger for more information.

Erasing Database-based Archiving

The upgrade removes tables and columns used for archiving from the database. The upgrade drops the following tables and columns:

- `bc_ArchiveAdminKey`
- `bc_ArchiveGraphRecord`
- `bc_ArchiveTransitionRec`
- `bc_ArchiveTypeKey`

The database archiving feature was not available for BillingCenter.

Removing InetSoft Reporting Support

The upgrade removes the following database elements that were involved in supporting InetSoft reporting:

- `bc_reportgroup`
- `bc_rolerptprivilege`
- `bc_rptgroupurprpt`
- `bc_sreereport`

The upgrade also drops the `bc_privilege` table, which is rebuilt later in the upgrade. Finally, the database upgrade removes the `reporting_admin` permission.

Adding NotificationSent to ProcessHistory

The upgrade creates a `NotificationSent` bit column on `ProcessHistory`. The upgrade sets the default value of `ProcessHistory.NotificationSent` for existing rows to `true`, so BillingCenter does not resend notifications.

Setting Parameter for Data Files Imported

The upgrade sets the `data_files_imported` parameter to `finished` in the `bc_Parameter` table, if the parameter is not already listed. This prevents rare issues with the upgrade caused by dependency on this parameter.

Dropping Extractable Columns

The upgrade removes the following columns from each entity that implements the `Extractable` delegate:

- `archiveid`
- `archivepartition`
- `extractready`
- `partition`

Not every `Extractable` entity includes these columns. The upgrade drops any of these columns that do exist on an `Extractable` entity.

This step is part of the removal of database-based archiving. The database archiving feature was not available for BillingCenter.

Setting IndividualStacks column on WorkQueueProfilerConfig to Non-nullable

The upgrade sets the `IndividualStacks` column on `bc_WorkQueueProfilerConfig` to non-nullable.

Changing Contact Foreign Keys on ContactAutoSyncWorkItem

This is a Guidewire platform-level upgrade step. Because BillingCenter does not use the Contact Auto Sync work queue, this step does not affect BillingCenter.

The upgrade first checks that the `bc_ContactAutoSyncWorkItem` table is empty. Then the upgrade drops the `minContactID` and `maxContactID` foreign keys to `bc_Contact` and replaces them with soft entity references `minContactRef` and `maxContactRef`.

Checking for Null Effective-dated Foreign Keys on EffDatedOnly Delegates

The upgrade checks that no effective-dated foreign keys on `effDatedOnly` delegates are null. If the upgrade finds any null effective-dated foreign keys on `effDatedOnly` delegates, it reports an error and stops the upgrade. The error includes an SQL query to identify the rows with issues.

Adding Subtype to WorkflowWorkItem

The upgrade adds a `Subtype` column to `cc_WorkflowWorkItem` with a default value of `WorkflowWorkItem`.

Renaming Primary Key Constraints and Indexes to Indicate Table Name

The upgrade renames primary key constraints and indexes to indicate the table name. For example, on Oracle, the upgrade renames the primary key index on `bc_Activity` to `PK_Activity`. On SQL Server, the upgrade renames the primary key index on `bc_Activity` to `bc_Activity_PK`.

Updating Columns to Support Very Large Data Sets

The upgrade changes the datatype of some columns in tables related to data distribution, data loading, and table statistics to be able to support very large data sets. In particular, on SQL Server the upgrade changes INT columns to BIGINT. For Oracle and SQL Server the upgrade changes DECIMAL columns to BIGINT for cases in which the column holds whole numbers.

This affects the following tables:

- bc_ArrayDataDist
- bc_ArraySizeCntDD
- bc_AssignableForKeyDataDist
- bc_AssignableForKeySizeCntDD
- bc_BeanVersionDataDist
- bc_BlobColDataDist
- bc_BooleanColDataDist
- bc_ClobColDataDist
- bc_DateAnalysisDataDist
- bc_DateBinnedDDDateBin
- bc_DateBinnedDDValue
- bc_ForKeyDataDist
- bc_GenericGroupCountDataDist
- bc_HourAnalysisDataDist
- bc_LoadInsertSelect
- bc_LoadOperation
- bc_LoadRowCount
- bc_NullableColumnDataDist
- bc_TableDataDist
- bc_TableUpdateStats
- bc_TypecodeCountDataDist
- bc_TypekeyDataDist

Dropping Staging Tables

The database upgrade drops the following staging tables:

- bcst_TmpABCCommission
- bcst_TmpChargeCmsnWriteoff
- bcst_TmpCPFP
- bcst_TmpInvStreamCreationOrder
- bcst_TmpItemStateCmsn
- bcst_TmpPayment
- bcst_TmpPolicy

The upgrade also drops the LoadCommandID column from the production table for each of these entities. These entities are not loadable as of BillingCenter 7.0.

This step applies to upgrades from versions prior to 7.0.1.

Converting Item Events from Retireable to Editable

The database upgrade checks that there are no retired `ItemEvent` records in `bc_ItemEvent`. If there are no retired `ItemEvent` records, the upgrade deletes the `bc_ItemEvent.Retired` column. If the upgrade detects retired `ItemEvent` records, it reports an error and stops the upgrade.

This step applies to upgrades from versions prior to 7.0.1.

Converting Line Items from Retireable to Editable

The database upgrade checks that there are no retired `LineItem` records in `bc_LineItem`. If there are no retired `LineItem` records, the upgrade deletes the `bc_LineItem.Retired` column. If the upgrade detects retired `LineItem` records, it reports the error and stops the upgrade.

This step applies to upgrades from versions prior to 7.0.1.

Adding ID Suffix to Delinquency Processing Columns

The database upgrade adds the suffix `ID` to the `LastDelinquencyProcessGrp` and `LastDelinquencyProcessUser` columns of the `bc_dynamic_assign` and `bc_group_assign` tables if it is not already present.

This step applies to upgrades from versions prior to 7.0.1.

Regenerating Work Item Tables

The database upgrade deletes all references to rows in the work item tables, including `bc_ProcessHistory`, `bc_InstrumentedWorker`, `bc_InstrumentedWorkerTask`, and then drops the tables. The tables are regenerated when BillingCenter starts and include any data model changes that Guidewire made to the work item tables between versions.

Populating Charge.WrittenDate and PolicyPeriod.TermConfirmed

The database upgrade adds the columns `bc_PolicyPeriod.TermConfirmed` and `bc_Charge.WrittenDate`. The upgrade sets `bc_PolicyPeriod.TermConfirmed` to true.

If the Charge belongs to a `BillingInstruction` with Subtype of `Audit`, `Cancellation`, `PolicyChange`, `PremiumReportBI`, or `Reinstatement`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `BillingInstruction` has a more recent `ModificationDate`. If the `BillingInstruction` has a `ModificationDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `ModificationDate` of the `BillingInstruction`.

If the Charge belongs to a `BillingInstruction` with Subtype of `BaseGeneral`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `PolicyPeriod` of the `BillingInstruction` has a more recent `PolicyPerEffDate`. If the `PolicyPeriod` of the `BillingInstruction` has a `PolicyPerEffDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `PolicyPerEffDate` of the `PolicyPeriod`.

If the Charge belongs to a `BillingInstruction` with Subtype of `Issuance`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `IssuancePolicyPeriod` of the `BillingInstruction` has a more recent `PolicyPerEffDate`. If the `IssuancePolicyPeriod` of the `BillingInstruction` has a `PolicyPerEffDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `PolicyPerEffDate` of the `IssuancePolicyPeriod`.

If the Charge belongs to a `BillingInstruction` with Subtype of `NewRenewal`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `NewRenewalPolicyPeriod` of the `BillingInstruction` has a more recent `PolicyPerEffDate`. If the `NewRenewalPolicyPeriod` of the `BillingInstruction` has a `PolicyPerEffDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `PolicyPerEffDate` of the `NewRenewalPolicyPeriod`.

If the Charge belongs to a BillingInstruction with Subtype of Renewal, the upgrade sets bc_Charge.WrittenDate to bc_Charge.ChargeDate unless the RenewalPolicyPeriod of the BillingInstruction has a more recent PolicyPerEffDate. If the RenewalPolicyPeriod of the BillingInstruction has a PolicyPerEffDate more recent than bc_Charge.ChargeDate, the upgrade sets bc_Charge.WrittenDate to the PolicyPerEffDate of the RenewalPolicyPeriod.

If the Charge belongs to a BillingInstruction with Subtype of Rewrite, the upgrade sets bc_Charge.WrittenDate to bc_Charge.ChargeDate unless the RewritePolicyPeriod of the BillingInstruction has a more recent PolicyPerEffDate. If the RewritePolicyPeriod of the BillingInstruction has a PolicyPerEffDate more recent than bc_Charge.ChargeDate, the upgrade sets bc_Charge.WrittenDate to the PolicyPerEffDate of the RewritePolicyPeriod.

If the Charge belongs to a BillingInstruction with Subtype of PremiumReportDueDate, the upgrade sets bc_Charge.WrittenDate to bc_Charge.ChargeDate unless the PremiumReportDDPolicyPeriod of the BillingInstruction has a more recent PolicyPerEffDate. If the PremiumReportDDPolicyPeriod of the BillingInstruction has a PolicyPerEffDate more recent than bc_Charge.ChargeDate, the upgrade sets bc_Charge.WrittenDate to the PolicyPerEffDate of the PremiumReportDDPolicyPeriod.

If the Charge belongs to a BillingInstruction with Subtype of AccountGeneral, CollateralBI, or SegregatedCollReqBI, the upgrade sets bc_Charge.WrittenDate to bc_Charge.ChargeDate.

This step applies to upgrades from versions prior to 7.0.1.

Resetting Upgrade Commission Work Item Table

The upgrade deletes bc_UpgradeCommissionWorkItem, the work item table for the Upgrade Commission work queue. The server rebuilds the bc_UpgradeCommissionWorkItem table afterwards.

This step applies to upgrades from versions prior to 7.0.1.

Dropping Temporary Tables Used for Previous Upgrades

The upgrade drops the following temporary tables and any corresponding staging tables, if they exist:

- FKTempTable
- NewIDsTempTable
- TmpChargeAcctGeneral
- TmpChargeCmsnWriteoff
- TmpChargeCommissionDup
- TmpChargeCommissionUnique
- TmpChargeGeneral
- TmpCmsnWrtffDlt
- TmpInvStreamCreationOrder
- TmpItemCommissionDup
- TmpItemCommissionUnique
- TmpOrigChargeCmsnConsol
- TmpPolicyCommissionDup
- TmpPolicyCommissionUnique
- TmpTAccountDup

These tables were used in previous upgrades and are no longer needed.

This step applies to upgrades from versions prior to 7.0.2.

Resetting Invoice Work Item Table

The upgrade deletes the `bc_InvoiceBilledWorkItem` work item table for the Invoice work queue. The server rebuilds the `bc_InvoiceBilledWorkItem` table afterwards.

This step applies to upgrades from versions prior to 7.0.2.

Resetting Invoice Due Work Item Table

The upgrade deletes the `bc_InvoiceDueWorkItem` work item table for the Invoice Due work queue. The server rebuilds the `bc_InvoiceDueWorkItem` table afterwards.

This step applies to upgrades from versions prior to 7.0.2.

Dropping PaymentComments from TmpDistItem

The upgrade drops the `bc_TmpDistItem.PaymentComments` column if it exists.

This step applies to upgrades from 7.0 versions prior to 7.0.3.

Populating Null PayableCriteria Columns on ChargeCommission

The upgrade populates any null `PayableCriteria` columns on `ChargeCommission` with the `PayableCriteria` value of the parent `CommissionSubPlan`.

This step applies to upgrades from versions prior to 7.0.3.

Populating Null PayableCriteria Columns on ItemCommission

The upgrade populates any null `PayableCriteria` columns on `ItemCommission` with the `PayableCriteria` value of the parent `ChargeCommission`.

This step applies to upgrades from versions prior to 7.0.3.

Converting Legacy CmsnTransferFromRollup Transactions

The upgrade removes any `CmsnTransferFromRollup` transactions that were created in BillingCenter 1.0, and replaces them with `ReserveCmsnEarned` and `PolicyCmsnPayable` transactions. The upgrade creates `ReserveCmsnEarned` and `PolicyCmsnPayable` transactions at the `ChargeCommission` level to duplicate the line items of `CmsnTransferFromRollup` at the `PolicyCommission` level. You could not create `CmsnTransferFromRollup` transactions in BillingCenter 2.0 or higher. So this step only does anything for databases that have previously been upgraded from BillingCenter 1.0 to 2.1 or 3.0 and still contain `CmsnTransferFromRollup` transactions.

The upgrade also removes the `CmsnTransferFromRollup` subtype from `Transaction`.

Before converting the transactions, the upgrade checks that there are no `CmsnTransferFromRollup` transactions that are reversed.

This step applies to upgrades from versions prior to 7.0.3.

Removing Legacy NegCmsnWriteoff Transactions

This step removes the legacy `NegCmsnWriteoff` transaction subtype. This transaction subtype has been deprecated since BillingCenter 2.0.

This step applies to upgrades from versions prior to 7.0.3.

Removing Legacy TimeBasedIncEarned Transactions

This step first checks that there are no legacy transactions of subtype TimeBasedIncEarned. If the upgrade does not detect such transactions, it removes the TimeBasedIncEarned transaction subtype. If the upgrade does detect such transactions, it reports an error and stops the upgrade.

This step applies to upgrades from versions prior to 7.0.3.

Dropping ItemEventCount from InvoiceItem

The upgrade drops the bc_InvoiceItem.ItemEventCount column.

This step applies to upgrades from versions prior to 7.0.3.

Adding and Populating AllInvoiceItemsExactlyPaid Bit Column to Invoice

The upgrade adds an AllInvoiceItemsExactlyPaid column to bc_Invoice. The upgrade sets AllInvoiceItemsExactlyPaid to true for an Invoice if all invoice items are exactly paid. The following conditions must apply:

- The Invoice subtype is StatementInvoice.
- The Invoice NetAmount minus NetAmountPaid minus NetAmountWrittenOff equals 0.
- For all InvoiceItem records for the Invoice, the PrimaryCommissionAmount minus PrimaryPaidCommission minus PrimaryCmsnPayableAmount minus PrimaryWrittenOffCommission equals 0.

This step applies to upgrades from versions prior to 7.0.4.

Adding GrossSettled Denormalized Bit Field To InvoiceItem

The upgrade adds a GrossSettled denormalized bit field to bc_InvoiceItem and sets the value of GrossSettled to true if Amount is equal to PaidAmount plus GrossAmountWrittenOff.

This step applies to upgrades from versions prior to 7.0.4.

Converting T-account Table from Retireable to Editable

The upgrade checks that there are no retired TAccount records in bc_TAccount. If there are no retired TAccount records, the upgrade deletes the bc_TAccount.Retired column. If the upgrade detects retired TAccount records, it reports an error and stops the upgrade.

This step applies to upgrades from versions prior to 7.0.4.

Resetting Work Item Tables

For upgrades from 7.0.1 through 7.0.3, the database upgrade drops and recreates the work item tables for the following work queues:

- Full Pay Discount
- Invoice Billed
- Invoice Due
- Payment Request

Resetting Funds Allotment Work Item Table

The upgrade drops and recreates the work item table for the Funds Allotment work queue.

This step applies to upgrades from versions prior to 7.0.4.

Adding FullyAllotted Bit Column to FundsTracker

The upgrade adds a denormalized bit column, `FullyAllotted` to `bc_FundsTracker`. The upgrade sets `FullyAllotted` to `true` if `AmountAllottedDenorm` equals `TotalAmount`.

This step applies to upgrades from versions prior to 7.0.4.

Adding HasChargeBilledTransaction and HasChargeDueTransaction to InvoiceItem

The upgrade adds `HasChargeBilledTransaction` and `HasChargeDueTransaction` columns to `bc_InvoiceItem`. The upgrade sets `HasChargeBilledTransaction` to `true` if a `ChargeBilled` exists for the `InvoiceItem`. The upgrade sets `HasChargeDueTransaction` to `true` if a `ChargeDue` transaction exists for the `InvoiceItem`.

This step applies to upgrades from versions prior to 7.0.4.

Updating Negative Write-off Reversals to be Loadable

The upgrade adds `LoadCommandID` and `ReversalDate` columns to `bc_NegativeWriteoffRev`. Both columns are created with null values. Negative write-off reversals can now be loaded using the `bcst_NegativeWriteoffRev` staging table.

Updating Negative Write-offs to be Loadable

The upgrade adds a `LoadCommandID` column to `bc_NegativeWriteoff`. The `LoadCommandID` column is created with a null value. Negative write-offs can now be loaded using the `bcst_NegativeWriteoff` staging table.

Updating T-account Balances

The upgrade updates `TAccount` balances, using the following steps:

- Creates a `TmpTAccount` temporary table to use during `BalanceDenorm` update of all `TAccounts`.
- Calculates the `TAccount.BalanceDenorm` field for each `TAccount` and inserts the value into `TmpTAccount`.
- Updates the `TAccount.BalanceDenorm` field for each `TAccount` using the value in `TmpTAccount`.

This step applies to upgrades from versions prior to 7.0.4.

Updating Charge TAccountContainer Balances

The upgrade recalculates the `BalanceDenorm` fields on all charge `TAccountContainers`.

This step applies to upgrades from versions prior to 7.0.4.

Updating `InvoiceItem.PrimaryCommissionAmount`

The upgrade updates the `bc_InvoiceItem.PrimaryCommissionAmount` denormalized field.

This step only runs for upgrades from versions prior to 3.0.6. This step may take a long time to run. If you are upgrading from a version prior to 3.0.8, Guidewire recommends that you first upgrade to 3.0.8 and run consistency checks before continuing the upgrade to 8.0. A direct upgrade from a 3.0 version prior to 3.0.8 can perform poorly, so for the best upgrade performance, upgrade to 3.0.8 first.

Regenerating `InstrumentedWorker` and Dropping `InstrumentedWorkerTask`

The database upgrade drops the `bc_InstrumentedWorker` and `bc_InstrumentedWorkerTask` tables. The `bc_InstrumentedWorker` table is regenerated when BillingCenter starts and includes data model changes that Guidewire made to the table between versions. The `bc_InstrumentedWorkerTask` table replaces the `bc_InstrumentedWorkerTask` table used in versions prior to BillingCenter 8.0.

Checking Uniqueness of Localized Admin Data

The upgrade checks that the `Name` value of the following entities is unique for that entity type:

- `AuthorityLimitProfile`
- `BusinessWeek`
- `Plan`
- `Region`
- `Role`

Dropping ClusterInfo

The upgrade drops the `bc_ClusterInfo` table. This table is renamed `bc_BatchServer`. The new table is created following the upgrade when BillingCenter starts. The `bc_BatchServer` table always contains only one row that describes the current batch server. This table is used by all cluster nodes to get the address of the current batch server and enforce that only a single batch server exists within the cluster.

BillingCenter 8.0 adds a `ClusterMemberData` entity that contains information about current cluster members. The information from this table is shown on the `Cluster Info` page. JGroups uses this table for the following:

- **JGroups over UDP** – Reporting and audit purposes. UDP multicast is used for discovery.
- **JGroups over TCP** – Reporting and discovery. JGroups reads the IP addresses of the current members from the `bc_ClusterMemberData` table.

Updating SpatialPoint on Address

The upgrade updates the `SpatialPoint` column on `bc_Address` with the data in the `Longitude` and `Latitude` columns. The upgrade checks for rows where `Longitude` or `Latitude` are non-null and the other is null. If the upgrade detects such rows, it reports an error, stops the upgrade, and provides an SQL query to find these rows.

After the upgrade updates `SpatialPoint`, it drops the `HTMID` column and the `Longitude` and `Latitude` columns.

Dropping Foreign Keys to ProcessHistory

The upgrade drops all `ProcessHistoryID` foreign keys that refer to `bc_ProcessHistory`.

Dropping WorkQueueName Column from WorkQueueWorkerControl

The upgrade drops the `bc_WorkQueueWorkerControl.WorkQueueName` column and deletes all current `WorkQueueWorkerControl` records. Later, the upgrade adds a new `LockName` column with unique index records.

Renaming WorkItem Column NumRetries to Attempts

The upgrade renames the `WorkItem` column `NumRetries` to `Attempts`.

Adding Subtype Column to Activity, Address, and History Tables

The upgrade adds a `Subtype` column to the `bc_Activity`, `bc_Address`, and `bc_History` tables. This allows Guidewire applications to create subtypes of these entities as needed.

Upgrading Consistency Check Tables

The upgrade makes the following changes to tables involved in consistency checks:

- drops the `NumThreads` and `Subtype` columns from `bc_dbConsistCheckRun`.
- drops the `Subtype` column from `bc_dbConsistCheckQueryExec`.

- updates null values of the `TableName` and `ThreadName` columns of `bc_dbConsistCheckQueryExec` to the value UNKNOWN.

Upgrading Database Statistics Tables

The upgrade makes the following changes to tables involved in gathering database statistics for BillingCenter:

- deletes all `bc_ProcessHistory` records for the Incremental Database Statistics process.
- drops the `Subtype` column from `bc_DatabaseUpdateStats`, `bc_TableUpdateStats`, and `bc_TableUpdateStatsStatement`.
- sets null values for `bc_TableUpdateStatsStatement.ObjectName` and `bc_TableUpdateStatsStatement.UpdateStatsStatement` to UNKNOWN.
- drops `bc_DatabaseUpdateStats.NumThreads`.
- drops `bc_TableUpdateStats.Deletes`, `bc_TableUpdateStats.Inserts` and `bc_TableUpdateStats.RowCnt`.

Dropping Upgrade-related Tables

The upgrade drops the following tables:

- `bc_UpgradeDBParameterPair`
- `bc_UpgradeDBParameterRow`
- `bc_UpgradeDBParameterSet`

Dropping AddressBookFingerprint from Contact and ContactCategoryScore

The upgrade drops the `AddressBookFingerprint` column from `bc_Contact` and `bc_ContactCatsScore`. The upgrade also drops the `AddressBookFingerprint` property from the `Contact` and `ContactCategoryScore` entities.

Dropping Obsolete Temporary Upgrade Tables

The upgrade drops the following obsolete temporary upgrade tables:

- `bc_TmpABCommission`
- `bc_TmpChargeAcctGeneral`
- `bc_TmpChargeGeneral`
- `bc_TmpChargePaidTxn`
- `bc_TmpCmsnReductionInvItem`
- `bc_TmpCPFP`
- `bc_TmpDist`
- `bc_TmpDistItem`
- `bc_TmpItemStateCmsn`
- `bc_TmpLineItem`
- `bc_TmpMoneyRcvd`
- `bc_TmpNegBldRevTrans`
- `bc_TmpPayment`
- `bc_TmpPolicy`
- `bc_UpgradeCollReq`

Removing Upgrade Commission Batch Process

The upgrade drops the `bc_UpgradeCommissionWorkItem` table as part of removing the Upgrade Commission batch process.

Dropping Columns

The upgrade drops the following columns:

- `bc_Charge.LegacyDepositOverride`
- `bc_InvoiceStream.Name`
- `bc_PolicyPeriod.ModNumber`

Updating Permissions

The upgrade updates permissions during the upgrade.

The upgrade converts some `feecreate` privileges to `gentxn` before removing remaining `feecreate` privileges. The upgrade updates `bc_Privilege` to update `Permission` from `feecreate` to `gentxn` if the `RoleID` is not associated with another privilege with permission `gentxn`.

The upgrade removes the following privileges:

- `acctdistedit`
- `acctdistview`
- `feecreate`

The upgrade adds the following privileges:

Privilege	Roles
<code>admindatachangeview</code>	<code>superuser</code>
<code>admindatachangeexec</code>	<code>superuser</code>
<code>wodatachangeedit</code>	<code>superuser</code>
<code>retpremplancreate</code>	<code>general_admin</code> <code>plan_admin</code> <code>superuser</code>
<code>retpremplanedit</code>	<code>general_admin</code> <code>plan_admin</code> <code>superuser</code>
<code>retpremplanview</code>	<code>billing_clerical</code> <code>billing_manager</code> <code>commissions_admin</code> <code>finance_manager</code> <code>general_admin</code> <code>plan_admin</code> <code>superuser</code> <code>underwriter</code>
<code>prodpmntedit</code>	<code>general_admin</code> <code>superuser</code>

Privilege	Roles
prodpromedit	general_admin
	superuser
prodpromview	general_admin
	superuser

Renaming Class Names of Transactions to Remove Abbreviations

The upgrade modifies class names of transactions to remove abbreviations, except for the word 'Transaction' itself.

Old Typekey	New Typekey
AgencyMoneyReceivedTxn	AgencyBillMoneyReceivedTxn
DirectBillMoneyRcvdTxn	DirectBillMoneyReceivedTxn
NegativeWriteoffAcct	AccountNegativeWriteoffTxn
CommissionReserved	CommissionsReserveTxn
CmsnReductionTxn	CommissionsReserveWriteoffTxn
CmsnPayableReduction	CommissionsReserveNegativeWriteoffTxn
CmsnRcvableReduction	CommissionsReservePositiveWriteoffTxn
NegWriteoffAcctContext	AccountNegativeWriteoffContext

The AccountNegativeWriteoffContext typekey is on bctl_AccountContext. The rest of the typekeys are on bctl_Transaction.

Adding Currency Columns

The upgrade adds silo currency columns to each siloed table and initializes these currency columns with the system default currency.

Upgrading CommissionWriteoffDistItem

The upgrade updates the class name, column names, and subtypes of the CommissionWriteoffDistItem entity. The upgrade makes the following changes:

- Renames the CmsnReductionID column of bc_ProducerContext to CommissionWriteoffDistItemID.
- Renames the bctl_ProducerContext typekey CmsnReductionContext to CommissionWriteoffDistItemContext.
- Renames the bctl_CmsnReductionType typekey payable to negative.
- Renames the bctl_CmsnReductionType typekey receivable to positive.
- Renames the bc_CmsnReduction column ReductionDate to DateWrittenOff.

Updating Subtypes on all ProducerPaymentSent Transaction Contexts

BillingCenter 7.0.4 adds a ProdPymtSentContext subtype on ProducerContext for ProducerPaymentSent transactions. This trigger updates the ProducerContext associated with all ProducerPaymentSent transactions to ProdPymtSentContext.

Converting Tables from Retireable to Editable

The upgrade converts the following tables from Retireable to Editable.

- bc_AccountContext
- bc_AgencyMoneyRcvdContext
- bc_AgencyDisbPaidContext
- bc_BillingInstruction
- bc_Charge
- bc_ChargePatternContext
- bc_ChargeProRataTx
- bc_CmsnsExpenseRollupCtx
- bc_CollateralContext
- bc_CreditContext
- bc_DBMoneyRcvdContext
- bc_NonReceivableItemCtx
- bc_SuspPymtContext
- bc_TAccountContainer
- bc_TransferTxContext

Ensuring No Retired Rows in Certain Tables

The upgrade checks that there are no retired rows in the bc_ChargeInstanceContext, bc_ProducerContext, or bc_Transaction tables. If the upgrade detects any retired rows in these tables, it reports an error and stops the upgrade.

Removing Unused Columns from TAccount

The upgrade removes the unused Description column from bc_TAccount. The upgrade also removes the Retired column, converting TAccount to an editable entity. The upgrade first checks that there are no retired rows in bc_TAccount. If the upgrade detects any retired bc_TAccount rows, it reports an error and stops the upgrade.

Updating Invoice Item for Exception Status Locking

The upgrade renames fields related to carrying forward exceptions on invoice items in 7.0 to the 8.0 exception lock versions.

Renames bc_InvoiceItem column PaymentCarriedForwardDate to PaymentExceptionLockDate.

Adds a PaymentExceptionLock typekey column to bc_InvoiceItem. The upgrade initializes PaymentExceptionLock to none. If the PaymentExceptionLockDate (formerly PaymentCarriedForwardDate) is not null, the upgrade sets PaymentExceptionLock to notexception.

Renames bc_InvoiceItem column PromiseCarriedForwardDate to PromiseExceptionLockDate.

Adds a PromiseExceptionLock typekey column to bc_InvoiceItem. The upgrade initializes PromiseExceptionLock to none. If the PromiseExceptionLockDate (formerly PromiseCarriedForwardDate) is not null, the upgrade sets PromiseExceptionLock to notexception.

Renaming BaseDist Column NetAmountDistributed to NetDistributedToInvoiceItems

The upgrade renames the bc_BaseDist column NetAmountDistributed to NetDistributedToInvoiceItems.

Updating BillingCenter for Change from State to Jurisdiction

All Guidewire InsuranceSuite applications have been updated to use `Jurisdiction` instead of `State`. The upgrade renames:

- `PolicyPeriod.RiskState` to `RiskJurisdiction`
- `CondCmsnSubPlan.AllStates` to `AllJurisdictions`
- `CondCmsnSubPlanState.State` to `Jurisdiction`
- `bc_CondCmsnSubplanState` to `bc_CondCmsnSubplanJurisdiction`

Checking References to Jurisdiction Typelist

The upgrade checks that the following references to the `Jurisdiction` typelist are valid:

- `bc_CommissionSubplan.AllJurisdictions`
- `bc_CondCmsnSubplanState.Jurisdiction`
- `bc_PolicyPeriod.RiskJurisdiction`

Moving AppliedDate to BaseMoneyReceived

The upgrade moves `AppliedDate` from `PaymentMoneyReceived` to `BaseMoneyReceived` and updates all promises that did not used to have the applied date to set their `AppliedDate` to the `ReceivedDate`.

Checking for Promises Linked to ProducerWriteoffs

The upgrade checks that there are no `Promises` linked to `ProducerWriteoffs`. If the upgrade detects `Promises` linked to `ProducerWriteoffs`, it reports an error and stops the upgrade.

Checking Payment Instruments of Agency Bill Payments

The upgrade checks that all agency bill payments with a zero amount have the producer unapplied payment instrument. The upgrade also checks that all agency bill payments with a non-zero amount do not have the producer unapplied payment instrument. If either condition is detected, the upgrade reports an error and stops.

Checking Payment Instruments of Direct Bill Payments

The upgrade checks that all direct bill payments with a zero amount have the account unapplied payment instrument. The upgrade also checks that all direct bill payments with a non-zero amount do not have the account unapplied payment instrument. If either condition is detected, the upgrade reports an error and stops.

Renaming AgencyCycleDist to AgencyCyclePaymentID on ProdWriteoffContext

The upgrade renames the `AgencyCycleDistID` column to `AgencyCyclePaymentID` on `bc_ProducerContext`.

Upgrading PromisedMoney

The upgrade moves the `MoneyBeingModified` edge link from the subclass `PaymentMoneyReceived` to the super-class `BaseMoneyReceived`. For each `PromisedMoney` marked as `Modified`, the upgrade attempts to determine which `PromisedMoney` it was modified into by finding a more recently created `PromisedMoney` attached to the same distribution. The upgrade then creates a link to the more recently created `PromisedMoney` in the `MoneyBeingModified` edge link table. This makes it so that promised monies are attached to the money they are modifying, just like any other money.

Upgrade the Distributed Amounts on Applied Promises

Trigger to upgrade the distributed amount on applied promises. In versions prior to 8.0, when a promise was applied, BillingCenter reversed the distributed amount down to zero. In 8.0, when viewing a promise that has been applied, you can see how much was distributed on the promise even if those amounts have been applied towards payment items.

Checking Amounts on Invoice Items

The upgrade checks that there are no positive amounts applied to negative invoice items or negative amounts applied to positive invoice items. The `netAmountOwed` and the `netAmountToApply` of a distribution must both be positive or both be negative. Similarly, the `CommissionAmountToApply` and `PrimaryCommissionAmount` must both be positive or both be negative. If the upgrade detects a distribution that does not meet this condition, it reports an error and stops.

Upgrading BillingLevel Typelist Values on Account

The upgrade sets all legacy `bc_Account` records that had `BillingLevel` set to `PolicyLevelBilling` to have `BillingLevel` of `PolicyDefaultUnapplied`. The upgrade sets all legacy `bc_Account` records that had `BillingLevel` set to `AccountBilling` to have `BillingLevel` of `Account`.

Wrapping Default Unapplied T-Accounts in UnappliedFunds

The upgrade wraps all default unapplied T-accounts with a new `UnappliedFund` entity. BillingCenter 8.0 uses the `UnappliedFund` entity to contain default unapplied T-accounts to:

- Provide database level protection. Queries can be against the `UnappliedFund` wrapper.
- Validate and protect at compile time.
- Prevent passing around of raw T-accounts with the potential of misuse.
- Provide a container for foreign keys to policy, reporting group, and others.
- Provide a container for methods like funds tracking methods and others.
- Make staging easier.
- Provide an entity on which users can put extension fields to enable linking the unapplied T-account to something other than a policy.

Making Work Item Tables Not Loadable

The upgrade drops the staging tables `bcst_AdvanceExpiration` and `bcst_ReleaseHoldsWorkItem`. The upgrade also drops the `LoadCommandID` column from the work item tables `bc_AdvanceExpiration` and `bc_ReleaseHoldsWorkItem`.

Upgrading NetDistributedToInvoiceItems

Trigger to upgrade the `NetDistributedToInvoiceItems` in `BaseDist`. Prior to 8.0, the `NetDistributedToInvoiceItems` includes both the `AmountDistributed` and all of the `SuspenseAmount`. In 8.0, the `NetDistributedToInvoiceItems` stores only the `AmountDistributed` and a new `NetInSuspense` attribute has been added to store the `SuspenseAmount`. The upgrade updates `NetDistributedToInvoiceItems` and add a new column for `NetInSuspense`.

Deleting AgencyCycleException

The upgrade updates the `AgencyCycleProcess` with exception comments from the `AgencyCycleException` entity. The upgrade then deletes `bc_AgencyCycleException` and `bct1_AgencyCycleException`.

Renaming ModifiedFromSPI to ModifiedFromBSDI

The upgrade renames `bc_ModifiedFromSPI` to `bc_ModifiedFromBSDI` to reflect moving the edge foreign key `ModifiedFrom` from `SuspensePaymentItem` to `BaseSuspDistItem`.

Dropping InvoiceID from BaseDist

The upgrade drops the `bc_BaseDist.InvoiceID` column.

Removing Edge Foreign Key from AgencyCyclePayment to AgencyCyclePromise

The upgrade removes the edge foreign key from `AgencyCyclePayment` to `AgencyCyclePromise` when the payment was reversed and promise unapplied. As of version 8.0, unapplying the promise when reversing the payment removes this link and sets the `AppliedDate` to `null`. Prior to 8.0, unapplying only set `AppliedDate` to `null`. This affected the `isApplied` method, which uses the existence of the link to a payment in 8.0.

Adding the MultiTAccountPattern Subtype

The upgrade updates the database to accommodate the `MultiTAccountPattern` subtype. The upgrade sets the `bc_TAccountPattern` columns `ChargeRollup` and `TAccountLazyLoaded` to nullable. These columns are only required for the `SingleTAccountPattern` subtype. The upgrade then changes the `Subtype` on `TAccountPattern` records to `SingleTAccountPattern`. The upgrade adds a new Designated Unapplied T-account pattern.

Setting Money Received for Reversal Transactions

The upgrade sets the foreign key on a money received context (`bc_AgencyMoneyRcvdContext` or `bc_DBMoneyRcvdContext`) to the appropriate money received for any money received reversal transactions. For `bc_AgencyMoneyRcvdContext`, the upgrade sets the `PaymentMoneyReceivedID` foreign key. For `bc_DBMoneyRcvdContext`, the upgrade sets the `DirectBillMoneyRcvdID` foreign key.

Adding Subtype Column to TAccount

The upgrade adds a `Subtype` column to `bc_TAccount`. In BillingCenter 8.0, the `TAccount` entity has a `MultiTAccount` subtype.

Creating Policy-level Billing Charge Patterns

The upgrade adds the `PolicyLateFee`, `PolicyPaymentReversalFee`, and `PolicyRecapture` charge patterns and associated T-account patterns. These charge patterns are for policy-level billing.

Adding UnappliedFund to Entities

The upgrade adds an `UnappliedFundID` foreign key column to the following entities:

- `AccountDisbursement`
- `AcctNegativeWriteoff`
- `Credit`
- `DirectBillMoneyRcvd`
- `FundsTracker`
- `FundsTransfer`
- `ZDBBMRReversal`
- `ZeroDollarDBMR`

The upgrade populates the `UnappliedFund` column with the default unapplied T-account.

Removing TransferTransaction Subtypes and Adding Foreign Keys to TransferTxContext

The upgrade converts all TransferTransaction subtypes (AcctProdTransfer, AcctsFundsTransferred, ProdAcctTransfer, ProducerFundsTransTxn) to TransferTransaction and drops the subtypes. The upgrade also adds SourceUnappliedTAccountID and TargetUnappliedTAccountID foreign keys to TransferTxContext. The upgrade then populates these columns.

Adding ReturnPremiumPlan

The upgrade adds the ReturnPremiumPlan subtype of Plan and links each PolicyPeriod to the ReturnPremiumPlan. A return premium plan governs how and when negative policy-level charges, otherwise known as returned premiums, are used to pay other charges in the system.

Converting Zero Dollar Agency Bill Payment Distributions to Credit Distributions

The upgrade converts zero dollar agency bill payment distributions to credit distributions. If the amount of the MoneyReceived is zero, then the upgrade sets the Subtype to ZeroDollarAMR. If the PaymentInstrument for the MoneyReceived has a PaymentMethod that is not Producer Unapplied, then the upgrade sets the PaymentInstrument for the MoneyReceived as follows:

If there are no PaymentInstruments with PaymentMethod of Producer Unapplied associated with the Producer of the MoneyReceived, the upgrade sets PaymentInstrument to a default payment instrument for Producer Unapplied.

If a producer is associated with the PaymentInstruments with PaymentMethod of Producer Unapplied, the upgrade sets PaymentInstrument to a PaymentInstrument associated with the producer. If there are multiple associated PaymentInstruments, then the one with the lowest ID is used.

The associated Transaction and LineItems are deleted in a later step.

Converting Non-zero ZeroDollarAMRs to AgencyBillMoneyRcvds

For bc_BaseMoneyReceived records that have Subtype of ZeroDollarAMR and an Amount that is not zero, the upgrade sets the Subtype to AgencyBillMoneyRcvd.

Converting ZeroDollarReversals to ZeroDollarDMRs

For bc_BaseMoneyReceived records that have Subtype of ZeroDollarReversal, the upgrade sets the Subtype to ZeroDollarDMR.

Converting Non-zero DirectBillMoneyRcvds to ZeroDollarDMRs

For bc_BaseMoneyReceived records that have Subtype of DirectBillMoneyRcvd and an Amount that is not zero, the upgrade sets the Subtype to ZeroDollarDMR.

Deleting Transactions and Line Items for Records with Zero Amount

Deletes Transactions and associated LineItems for DirectBillMoneyReceivedTxn and AgencyBillMoneyReceivedTxn records with an amount of zero.

Converting Non-zero ZeroDollarDMRs to DirectBillMoneyRcvds

For bc_BaseMoneyReceived records that have Subtype of ZeroDollarDMR and an Amount that is not zero, the upgrade sets the Subtype to DirectBillMoneyRcvd.

Renaming FundsSourceType and FundsUseType Account Columns

The upgrade renames `bc_FundsSourceType.Account` to `bc_FundsSourceType.UnappliedFund` and `bc_FundsUseType.Account` to `bc_FundsUseType.UnappliedFund`. Both typekeys are used for unapplied fund balance forward funds trackers.

Dropping AccountID from FundsTracker

The upgrade drops the column `bc_FundsTracker.AccountID`.

Moving ReportingGroupID Foreign Key from Account to UnappliedFunds

The upgrade moves the `ReportingGroupID` foreign key from `bc_Account` to `bc_UnappliedFunds`.

Updating General BillingInstruction Records

In 8.0, the supertype of `General BillingInstruction` has changed from `BaseGeneral` to `ExistingPlcyPeriodBI`, so `General` uses different names for the same data now. The upgrade makes the following updates to `bc_BillingInstruction` records with Subtype of `General`:

- Sets the `ModificationDate` to the `BillingInstructionDate`.
- Sets the `AssociatedPolicyPeriodID` to the `PolicyID`.
- Sets the `BillingInstructionDate` to `null`.
- Sets the `PolicyID` to `null`.

Moving PolicyPeriodID to NewPolicyPeriodID for NewPlcyPeriodBI Subtypes

In 8.0, the individual `PolicyPeriodID` fields on `NewPlcyPeriodBI` subtypes were rolled up into a `NewPolicyPeriodID` field. The upgrade moves the data into the new column. This affects the following subtypes:

- Issuance
- NewRenewal
- Renewal
- Rewrite

Adding PCPublicID to Policy

The upgrade adds nullable column `PCPublicID` to `bc_Policy`. This column is used for integration with PolicyCenter.

Dropping DistributedDenorm from BaseMoneyReceived

The upgrade drops the `DistributedDenorm` column from `bc_BaseMoneyReceived`.

Dropping RenewalNumber from PolicyPeriod

The upgrade drops the `RenewalNumber` column from `bc_PolicyPeriod`.

Dropping ReceivableAgingWorkItem

The upgrade checks that there are no work items in the `bc_ReceivableAgingWorkItem` table. If there are no work items, the upgrade drops the `bc_ReceivableAgingWorkItem` table.

Dropping ReceivableAging

The upgrade drops `bc_ReceivableAging`.

Adding Producer Code to AgencyPaymentItem

The upgrade adds the producer code to `AgencyPaymentItem` if it can find a related producer code.

Populating Currency for All Monetary Amounts

The upgrade populates the `Currency` field of all `MonetaryAmount` records in the system. The upgrade sets the value of the column to the default currency.

Renaming Deferred Upgrade Batch Process

The upgrade renames the `DeferredUpgrade` batch process type to `DeferredUpgradeTasks`.

Truncating bc_Dynamic_Assign

The upgrade truncates the `bc_Dynamic_Assign` table.

Dropping bc_t1_Template

The upgrade drops the `bc_t1_Template` table.

Dropping Columns from WorkItem Tables

The upgrade drops the `AvailableSince` and `LastUpdateTime` columns from all `bc_WorkItem` tables.

Upgrading Shared Typekey Data

The upgrade checks for subtypes with typekeys that have the same field name, different column names, and only one column exists in the database. If any such records exist, the upgrade moves the data to the correct column.

Renaming LOBCode Typekey

The upgrade renames the `LOBCode` typekey `HomeOwners` to `Homeowners`.

Dropping DunningInterval from Plan

The upgrade drops the `bc_Plan.DunningInterval` column.

Adding ListBillAccountExcessTreatment to ReturnPremiumPlan

The upgrade adds a `ListBillAccountExcessTreatment` column to all `ReturnPremiumPlan` instances with value of `POLICY_PAYER_UNAPPLIED`.

Adding PaymentAllocationPlans

The upgrade adds `PaymentAllocationPlans` representing each distribution limit and links to each `Account` according to the previous `DistributionLimitType` of the `Account`.

Updating Denormalized Fields on InvoiceItem

The upgrade updates denormalized fields `CanBePaidMoreByAgencyBill` and `CanBePromisedMoreByAgencyBill` on `bc_InvoiceItem` to indicate whether or not each invoice item can be paid or promised any more with agency billing.

Associating Users with User-specific Custom Authority Limit Profiles

The upgrade adds records to `bc_CustomALPUser` to associate users with user-specific custom authority limit profiles and deletes `bc_AuthorityLimitProfile.Custom` if it exists.

Creating and Updating AccountContext.UnappliedFundID

The upgrade creates and updates the `bc_AccountContext.UnappliedFundID` foreign key. For `AccountContext` records, the upgrade sets the `UnappliedFundID` to the `DefaultUnappliedFund` of the associated `Account`. For `DisbPaidContext` records, the upgrade sets the `UnappliedFundID` to the `UnappliedFund` of the associated `AccountDisbursement`. For `AccountNegativeWriteoffContext` records, the upgrade sets the `UnappliedFundID` to the `UnappliedFund` of the associated `AcctNegativeWriteoff`.

Adding Payment Allocation Privileges Included with BillingCenter 8.0.1

The upgrade adds the following payment allocation privileges:

Privilege	Roles
<code>payallocplancreate</code>	<code>general_admin</code> <code>plan_admin</code> <code>superuser</code>
<code>payallocplanedit</code>	<code>general_admin</code> <code>plan_admin</code> <code>superuser</code>
<code>payallocplanview</code>	<code>billing_clerical</code> <code>billing_manager</code> <code>commissions_admin</code> <code>finance_manager</code> <code>general_admin</code> <code>plan_admin</code> <code>superuser</code> <code>underwriter</code>

Adding Direct Collector Role

The upgrade adds the Direct Collector role.

Dropping PolicyDlnqProcessID from LegacyDlnqWorkItem

The upgrade drops the `bc_LegacyDlnqWorkItem.PolicyDlnqProcessID` column.

Viewing Detailed Database Upgrade Information

BillingCenter includes an [Upgrade Info](#) page that provides detailed information about the database upgrade. The [Upgrade Info](#) page includes information on the following:

- version numbers before and after the database upgrade
- configuration parameters used during the database upgrade
- SQL queries for version checks that test if the database is in condition to be upgraded
- changes made to specific tables, including which version triggers modified the table or its data and the SQL statement executed to make each change

- version triggers that the upgrade ran, including which tables the trigger ran against, a description, the SQL statement run against each table and the start and end time
- a list of upgrade steps, including the table on which the step operated
- a table registry including table IDs before and after upgrade

The database upgrade deletes upgrade instrumentation information for prior database upgrades. If the database upgrade detects any prior upgrade instrumentation data, it reports a warning and deletes the data. If you have run previous database upgrades, and you want to preserve upgrade instrumentation details, download this information.

To download upgrade instrumentation details

1. Start the BillingCenter server if it is not already running.
2. Log in to BillingCenter with the superuser account.
3. Press ALT+SHIFT+T to access **System Tools**.
4. Click **Info Pages**.
5. Select **Upgrade Info** from the **Info Pages** drop-down.
6. Click **Download** to download a ZIP file containing the detailed upgrade information.

Dropping Unused Columns on Oracle

By default, the BillingCenter database upgrade on Oracle marks columns that have been removed from the data model as unused. Marking a column unused is a faster operation than dropping a column. Because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments.

You can configure the upgrade to drop removed columns immediately by setting the `deferDropColumns` parameter to `false` before running the database upgrade. This parameter is within the `<upgrade>` block of the `<database>` block of `database-config.xml`.

If you did not set `deferDropColumns` to `true` before the upgrade, perform the procedure in this topic to drop unused columns after the upgrade. You can drop the unused columns after the upgrade during off-peak hours to free the space. BillingCenter does not have to be shutdown to perform this maintenance task. You can drop all unused columns in one procedure, or you can drop unused columns for individual tables.

To drop all unused columns

1. Create the following Oracle procedure to purge all unused columns:

```
DECLARE
    dropstr VARCHAR2(100);
    CURSOR unusedcol IS
        SELECT table_name
        FROM user_unused_col_tabs;
BEGIN
    FOR tabs IN unusedcol LOOP
        dropstr := 'alter table '
            || tabs.table_name
            ||' drop unused columns';
        EXECUTE IMMEDIATE dropstr;
    END LOOP;
END;
```

2. Run the procedure during a period of relatively low activity.

To drop unused columns for a single table (or all tables)

1. Start the server to run the schema verifier. The schema verifier runs each time the server starts. If there are unused columns, the schema verifier reports a difference between the physical database and the data model. The schema verifier reports the name of each table and provides an SQL command to remove unused columns from each table.
2. Run the SQL command provided by the schema verifier. This command has the following format:

```
ALTER TABLE tableName DROP UNUSED COLUMNS
```

Exporting Administration Data for Testing

Guidewire recommends that you create a small set of administration data from an upgraded data set. Use this data for development and testing of rules and libraries with BillingCenter 8.0.4. This procedure is optional.

You might have already created an upgraded administration data set by following the procedure “Upgrading Administration Data for Testing” on page 134. If you followed that procedure, or you do not want an administration-only data set for testing purposes, you can skip this topic.

To create an administration data set for testing

1. Export administration data from your upgraded production database.
 - a. Start the BillingCenter 8.0.4 server by navigating to `BillingCenter/bin` and running the following command:
`gwbc dev-start`
 - b. Open a browser to BillingCenter 8.0.4.
 - c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
 - d. Click the **Administration** tab.
 - e. Click → Utilities → Export Data.
 - f. Select the **Admin** data set to export.
 - g. Click **Export** to download the `admin.xml` file.
2. Create a new database account for the development environment on a database management system supported by BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <http://guidewire.custhelp.com>.
See “Configuring the Database” on page 23 in the *Installation Guide* for instructions to configure the database account.
3. Install a new BillingCenter 8.0.4 development environment. Connect this development environment to the new database account that you created in step 2. See the *BillingCenter Installation Guide* for instructions.
4. Copy the `admin.xml` file that you exported to a location accessible from the new development environment.
5. Create an empty version of `importfiles.txt` in the `modules/configuration/config/import/gen` directory of the new development environment.
6. Create empty versions of the following CSV files:
 - `activity-patterns.csv`
 - `authority-limits.csv`
 - `reportgroups.csv`
 - `roleprivileges.csv`

- `rolereportprivileges.csv`

Leave `roles.csv` as the original complete file.

7. Import the administration data into the new database:

- a. Start the BillingCenter 8.0.4 development server by navigating to `BillingCenter/bin` and running the following command:

```
gwbc dev-start
```

- b. Open a browser to BillingCenter 8.0.4.

- c. Log on as a user with the `viewadmin` and `soapadmin` permissions.

- d. Click the **Administration** tab.

- e. Click → Utilities → Import Data.

- f. Click **Browse....**

- g. Select the `admin.xml` file that you exported from the upgraded production database and modified.

- h. Click **Open**.

Upgrading Phone Numbers

BillingCenter 8.0 has a different format for phone numbers. Each phone number type has two additional fields in 8.0: a country code and an extension. The country code is a typekey to the `PhoneCountryCode` typelist, which is a list of regions and their regional phone codes.

BillingCenter 8.0 provides a plugin interface, `IPhoneNormalizerPlugin`, that you can use to customize the conversion of legacy phone numbers to the 8.0 standard. The interface includes the following method signatures:

- `isPossibleNumber(String) : boolean`
- `isPossibleNumberWithExtension(String) : boolean`
- `normalizeNumberIfPossible(String) : String`
- `parsePhoneNumber(String) : GWPhoneNumber`
- `formatPhoneNumber(GWPhoneNumber number) : String`
- `normalizePhoneNumbersInBean(KeyableBean) : void`
- `normalizePhoneNumbersInArchive(IArchivedEntity, java.util.List<PhoneColumnProperties>) : void`

The plugin is called by the Phone Number Normalizer work queue. The plugin is also called whenever an entity containing a phone number is modified in BillingCenter.

Guidewire provides a default implementation of the plugin, `gw.api.phone.DefaultPhoneNormalizerPlugin`. If you disabled the phone number input mask or imported phone numbers, you might need to customize the plugin implementation. If you added new phone fields on existing objects or extension objects, customize or extend the plugin implementation to handle the additional phone fields. The `gw.api.util.PhoneUtil` class contains helper methods to facilitate formatting and parsing phone number records.

Define both `isPossibleNumber` and `isPossibleNumberWithExtension` methods to be very loose, non-country specific, validations. These methods essentially just need to check if the passed string could be a number in any country.

In the default phone normalizer plugin implementation, the `normalizeNumberIfPossible` method strips all decorator and formatting characters from the number if `isPossibleNumber` returns `true`. The normalizer ignores all numeric characters as well as + and * characters.

The `parsePhoneNumber` and `formatPhoneNumber` methods are used to convert between BillingCenter 7.0 and BillingCenter 8.0 phone numbers. The `parsePhoneNumber` method parses a string into a `GWPhoneNumber` object if possible. `GWPhoneNumber` is an interface that defines a standard BillingCenter 8.0 phone number object. See the Javadoc for further details. The `parsePhoneNumber` method is for converting phone numbers from versions prior to 8.0 to the 8.0 standard. The `formatPhoneNumber` method formats a `GWPhoneNumber` object into a single string. The `formatPhoneNumber` method is for converting 8.0 phone numbers to the 7.0 standard.

The plugin only normalizes a phone number if `isPossibleNumber` returns `true`. If `isPossibleNumber` returns `true`, the plugin calls `parsePhoneNumber` to convert the number to a `GWPhoneNumber` object.

By default, the maximum length of a phone number extension field is four. You can change the maximum length of phone number extensions by specifying an `extensionLength` parameter on the plugin implementation.

1. In Studio, open configuration → config → Plugins → registry → `IPhoneNormalizerPlugin.gwp`.
2. Click the Add Parameter  icon next to Parameters.
3. Enter `extensionLength` for the key.
4. Enter a numeric value for value.

You can call the phone normalizer plugin when adding a contact record from an external system to convert the phone number to the BillingCenter 8.0 standard. You might need to customize the plugin depending on the format of your source data.

The Phone Number Normalizer work queue generates work items for phone numbers with a country code of `unparseable` or `null`, indicating that the plugin has not yet processed the number.

If you are using ContactManager, run the Phone Number Normalizer work queue for ContactManager first. Then run the Phone Number Normalizer work queue for BillingCenter. Phone numbers in BillingCenter may become out of sync with ContactManager while the ContactManager Phone Number Normalizer work queue is running. It is safe to sync contacts in BillingCenter that become out of sync with ContactManager. When you run the Phone Number Normalizer work queue for BillingCenter, it skips the previously synced records.

Eventually, run the Phone Number Normalizer work queue for all of your Guidewire applications.

For performance reasons, run the Phone Number Normalizer work queue at off-peak hours. Some functionality, such as ContactManager's de-duplication feature could perform poorly while the Phone Number Normalizer work queue runs. You could see Concurrent Data Change Exceptions if you modify an existing contact at the same time as the Phone Number Normalizer work queue. If this occurs, reload the contact and attempt the update again.

Final Steps After The Database Upgrade is Complete

This section describes the procedures to run after you have completed the upgrade procedure and migration of configurations and integrations. The processes in this section provide you with a benchmark of the new system. Completing these steps is particularly important to going live in a production environment.

Use these procedures to revalidate the database:

- “Validating the Database Schema” on page 137
- “Checking Database Consistency” on page 138, including “Checking that Contacts Have Unique Addresses” on page 187
- “Creating a Data Distribution Report” on page 138
- “Generating Database Statistics” on page 139. You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded BillingCenter in a production environment.

- “Backing up the Database After Upgrade” on page 187

Completing Deferred Upgrade

If you have archiving enabled, and you did not set `deferCreateArchiveIndexes` to `false`, run the `Deferred Upgrade Tasks` batch process as soon as possible after the completion of the upgrade. To run the `Deferred Upgrade Tasks` batch process, use the `admin/bin/maintenance_tools` command:

```
maintenance_tools -password password -startprocess deferredupgradetasks
```

Reenabling Database Logging

You might have disabled logging of direct insert and create index operations during the database upgrade. After you complete the database upgrade successfully, you can reenable logging by setting `allowUnloggedOperations` to `false` in the `<upgrade>` block. For example:

```
<database ...>
...
<upgrade allowUnloggedOperations="false">
...
</upgrade>
</database>
```

For SQL Server, if you changed the recovery model from Full to Simple or Bulk logged during the upgrade, you can revert the recovery model. If you deferred migrating to 64-bit IDs, you might disable logging again when you perform the migration.

Checking that Contacts Have Unique Addresses

An `Address` cannot be shared by more than one `Contact`. BillingCenter 8.0 includes a commit-time check that does not allow a shared reference to an address instance even when one of the referring `Contact` or `ContactAddress` instances is retired. If you have multiple contacts at the same address, you can create separate address instances with the same field values.

A database consistency check on the `Contact` entity reports an error if it detects multiple `Contact` records using the same `PrimaryAddress`.

Before using BillingCenter 8.0.4 in production, run database consistency checks to find any instances of shared references to address instances. If the consistency check reports shared addresses, contact Guidewire Support for assistance fixing your database.

Backing up the Database After Upgrade

Finally, before going live, back up the upgraded database. This provides you with a snapshot of the initial upgraded data set, if an unanticipated event occurs just after going live.

Upgrading BillingCenter from 7.0.x for ContactManager

This topic lists the manual tasks required to upgrade BillingCenter 7.0.x and ContactManager 7.0.x to BillingCenter 8.0.x and ContactManager 8.0.x. Before starting this upgrade process, you must have run the Guidewire upgrade and merge tools. Additionally, Guidewire recommends that you first upgrade ContactManager, integrate BillingCenter with ContactManager, and refresh the ContactManager web APIs.

See also

- “Upgrading the BillingCenter 7.0.x Configuration” on page 95
- “Upgrading the BillingCenter 7.0.x Database” on page 133

This topic includes:

- “Configuration File Changes in BillingCenter” on page 190
- “Manually Upgrading BillingCenter to Integrate with ContactManager” on page 191

Configuration File Changes in BillingCenter

The following table shows the files and classes used to configure BillingCenter to work with ContactManager 7.0.x and the files that replace them in BillingCenter 8.0.x. You can use this table as a reference for the list of files you see in the configuration upgrade tool.

BillingCenter 7.0 Name	BillingCenter 8.0 Name
IContactSystemPlugin Name of both plugin registry, IContactSystemPlugin.xml, and plugin interface, IContactSystemPlugin.java	ContactSystemPlugin Name of both the plugin registry, ContactSystemPlugin.gwp, and the plugin interface, ContactSystemPlugin.java See “Integrating ContactManager with BillingCenter in QuickStart” on page 60 in the <i>Contact Management Guide</i> .
ContactManagerSystemPlugin Plugin class that implements IContactSystemPlugin.java. Package name – gw.plugin.contact.ab700	ABContactSystemPlugin (ab800 version) Plugin class that implements ContactSystemPlugin.java. Package name – gw.plugin.contact.ab800. When ContactManager 8.0 is installed with PolicyCenter 8.0, register this plugin implementation.
ContactIntegrationXMLMapper.gs Name of the class that maps contact data as XML between BillingCenter and ContactManager. Package name – gw.plugin.contact.impl	ContactManagerSystemPlugin (ab700 version) Plugin class that implements IContactSystemPlugin.java. Package name – gw.plugin.contact.ab700. When ContactManager 7.0 is installed with BillingCenter 8.0, register this plugin implementation in the registry ContactSystemPlugin.gwp. Note: Plugin implementation class name and plugin interface it implements are different from the ab800 version.
bc-to-cm-type-mapping.xml and cm-to-bc-type-mapping.xml Name of the files used to specify how to map differing entity names and typecodes between BillingCenter and ContactManager.	ContactMapper.gs This XML mapping class has been completely changed in BillingCenter 8.0. It supports integration with ContactManager 8.0. Package name – gw.contactmapper.ab800 See “Core Application ContactMapper Class” on page 246 in the <i>Contact Management Guide</i> .
	ContactIntegrationXMLMapper.gs This XML mapping class supports integration with ContactManager 7.0. Package name – gw.contactmapper.ab700
	BCNameMapper.gs This Gosu class replaces the two XML mapping files in BillingCenter 8.0. Package name – gw.contactmapper.ab800 See “Core Application Mapping” on page 141 in the <i>Contact Management Guide</i> .

BillingCenter 7.0	BillingCenter 8.0
Name	Name
ContactAPI.gs The web service that implements ABCClientAPI to provide a way for ContactManager to call into BillingCenter. Package name – gw.webservice.bc.bc700.contact	ContactAPI.gs (ab800 version) Package name – gw.webservice.bc.bc800.contact ContactAPI.gs (ab700 version) This web service supports integration with ContactManager 7.0 and has been moved to a different package. Package name – gw.webservice.bc.bc700.contact

Manually Upgrading BillingCenter to Integrate with ContactManager

This topic describes tasks you might have to perform to complete a BillingCenter 7.0.x upgrade when you have ContactCenter installed.

Prior to performing the tasks in this topic, do the following:

1. Run the Configuration Upgrade tool and perform the automatic upgrade for the BillingCenter configuration. See “Upgrading the BillingCenter 7.0.x Configuration” on page 95. Do not make changes yet to the files listed in this topic for BillingCenter. You make those changes later as described in this topic.
2. Run the Database Upgrade tool to upgrade for the BillingCenter database. See “Upgrading the BillingCenter 7.0.x Database” on page 133.
3. You can perform any manual configuration upgrades except those related to files listed later in this topic. Before making those changes, wait until you configure ContactManager, regenerate its SOAP API, and refresh that API in BillingCenter Studio, as described in the steps that follow.
4. Manually configure ContactManager. See “Upgrading ContactManager from 7.0.x” on page 193.
5. Integrate BillingCenter and ContactManager as described at “Integrating ContactManager with Guidewire Core Applications” on page 45 in the *Contact Management Guide*.

Mapping Your Contact Extensions

If you have made extensions to the Contact entity, such as new subtypes or fields, the files you use to map these extensions to ContactManager have changed and require updating. To update the new files:

1. Examine the BillingCenter 7.0.x files `bc-to-ab-data-mapping.xml` and `ab-to-bc-data-mapping.xml`. In BillingCenter 7.0.x Studio in the Resources pane, navigate to `configuration → Other Resources`, and then click each file to open it in an editor.
2. Update the Gosu class `BCNameMapper` with the names of any extended subentities and any typelists whose names differ between BillingCenter and ContactManager. In BillingCenter 8.0.x Studio press `CTRL+N` and enter `BCNameMapper` to find this class, and then double-click the class name to open it in the editor.
3. As described in “Configuration File Changes in BillingCenter” on page 190, the BillingCenter 7.0 contact XML mapping file `ContactIntegrationXMLMapper` is now called `ContactMapper`. If you have customized `ContactIntegrationXMLMapper`, you will need to port your code to `ContactMapper`, which has been completely refactored to provide:
 - Simpler code for adding foreign keys and array references
 - Ability to specify fields that determine if a contact is in sync

- Ability to specify fields for a contact that are persisted in the core application

For a description of this class, see “ContactMapper Class” on page 243 in the *Contact Management Guide*.

- As described in “Configuration File Changes in BillingCenter” on page 190, the `ClaimCenter contact-sync-config.xml` file is no longer being used. Its functionality has been replaced by the `withAffectsSync` method in `ContactMapper` and the `RelationshipSyncConfig` class. If you use `contact-sync-config.xml` to exclude fields or relationships, you must port your settings.

See “Synchronizing ClaimCenter Contact Fields” on page 193 in the *Contact Management Guide*.

See also

- “Working with Contact Mapping Files” on page 140 in the *Contact Management Guide*
- “Core Application Mapping” on page 141 in the *Contact Management Guide*

Parameter `transactionId` Removed from ContactManager Web Services

As shown in the following table, the ContactManager web services `ABCClientAPI` and `ABContactAPI` no longer use the `transactionId` parameter. If you have written code that calls these web services, you must rewrite it.

ContactManager 7.0	ContactManager 8.0
<p><code>ABCClientAPI</code> and <code>ABContactAPI</code> methods have a <code>transactionId</code> parameter.</p> <p>The following methods used a <code>transactionId</code> parameter to identify the method call to the web service:</p> <ul style="list-style-type: none"> <code>ABCClientAPI</code> <ul style="list-style-type: none"> <code>.mergeContacts</code> <code>.removeContact</code> <code>.updateContact</code> <code>ABContactAPI</code> <ul style="list-style-type: none"> <code>.createContact</code> <code>.removeContact</code> <code>.updateContact</code> 	<p><code>ABCClientAPI</code> and <code>ABContactAPI</code> methods no longer use a <code>transactionId</code> parameter.</p> <p>The transaction ID is now set for the SAOP header in <code>gw.webservice.contactapi.ContactAPIUtil.setTransactionId</code></p>

Instead of passing the transaction ID as part of the contact method call, it is set in a separate method. For example:

```
ContactAPIUtil.setTransactionId(
    ABContactAPI.Config,
    transactionId)
ABContactAPI.updateContact(xml)
```

See also

- “Setting Guidewire Transaction IDs” on page 75 in the *Integration Guide*
- “`ABCClientAPI` Interface” on page 240 in the *Contact Management Guide*
- “`ABContactAPI` Web Service” on page 233 in the *Contact Management Guide*

Upgrading ContactManager from 7.0.x

This topic covers the manual steps needed to perform an upgrade of ContactManager 7.0x to ContactManager 8.0.

This topic includes:

- “Database Upgrade Steps in ContactManager” on page 193
- “Configuration File Changes in ContactManager” on page 194

Database Upgrade Steps in ContactManager

The database upgrade for ContactManager is the same as that for BillingCenter, except for the one manual step described in this topic. For information on preparing for database upgrade, see “Upgrading the BillingCenter 7.0.x Database” on page 133.

Preserving MatchSetKey Column Data

The database upgrade by default drops the `MatchSetKey` column for `ABContact` and any subentities of `ABContact`. The base configuration of ContactManager 8.0 has the `MatchSetKey` column commented out in `ABContact.etx`. If you have data in the `MatchSetKey` column in ContactCenter 7.0 that you want to preserve, before starting the database upgrade, uncomment this column in `ABContact.etx` in ContactManager 8.0.

To uncomment the column

1. Open the ContactManager 8.0 file `ABContact.etx` in a text editor.

The file is located in the folder `ContactManager/modules/configuration/config/extensions/entity`, where `ContactManager` is your main ContactManager 8.0 installation directory.

2. Remove the comments surrounding the `MatchSetKey` column definition.
3. Save the file.

Ensuring that LinkID Is Unique

If you are upgrading from a ContactManager version prior to 7.0.6, you must perform the database operation described in this topic. If your ContactManager version is 7.0.6 or later, the upgrade automatically detects and resolves duplicate LinkID fields for you.

Before performing the database upgrade, you must ensure that all contact LinkID fields in your ContactManager 7.0 database are unique. Contact Guidewire Support for the query to run on your database and the process for updating any duplicate LinkID fields.

Configuration File Changes in ContactManager

This topic covers the manual steps needed to perform a configuration upgrade of ContactManager 7.0x to ContactManager 8.0. Prior to performing these upgrade steps, you must run the upgrade software and perform automatic upgrades.

The following table shows files and classes used to configure ContactManager 7.0.x to work with BillingCenter and the corresponding ContactManager 8.0 files that replace them. You can use this table as a reference for the list of files you see in the configuration upgrade tool.

The following classes and XML files have either new names or new Gosu classes, as appropriate.

Because ContactManager has changed a number of the files used to integrate with the Guidewire applications, it is likely that you will need to manually update configuration files. In particular, you will need to make manual updates:

- If you have made changes to the ABContact data model.
- If you have changed any of the files that are listed in “Configuration File Changes in ContactManager” on page 194.

In general, the steps for upgrading ContactManager are:

1. Run the configuration upgrade tool and perform an automatic upgrade of the ContactManager configuration. See “Upgrading the BillingCenter 7.0.x Configuration” on page 95.
2. Run the database upgrade tool to upgrade the ContactManager database. See “Upgrading the BillingCenter 7.0.x Database” on page 133.
3. Manually configure files in ContactManager—the subject of this topic.
4. Upgrade BillingCenter as described at “Upgrading BillingCenter from 7.0.x for ContactManager” on page 189.

Manually Configuring Changed Files

If you have customized any of the classes described in the table at “Configuration File Changes in ContactManager” on page 194, you must reapply your changes to each class. This topic provides some additional information about some of the classes that have changed.

ContactManager 7.0	ContactManager 8.0
ABContactAPI.gs The web service available to core applications to make contact related calls into ContactManager, such as create, retrieve, update, and delete contacts. Package name – gw.webservice.ab.ab700.abcontactapi.	ABContactAPI.gs This web service has been updated to support services and pending contact changes. Package name – gw.webservice.ab.ab800.abcontactapi To find and open the class in Studio, use the Class Search dialog in non-project mode. Note: Press CTRL+N twice to turn on non-project class searching. See “ABContactAPI Web Service” on page 233 in the <i>Contact Management Guide</i> .
ABCClientAPI and ABContactAPI methods have TransactionID parameter. The following methods used a transactionID parameter to identify the method call to the web service: <ul style="list-style-type: none">• ABCClientAPI<ul style="list-style-type: none">.mergeContacts.removeContact.updateContact• ABContactAPI<ul style="list-style-type: none">.createContact.removeContact.updateContact	ABCClientAPI and ABContactAPI methods no longer use TransactionID parameter. The transaction ID is now set for the SAOP header in gw.webservice.contactapi.ContactAPIUtil.setTransactionId Instead of passing the transaction ID as part of the contact method call, it is set in a separate method. For example: ContactAPIUtil.setTransactionId(ABContactAPI.Config, transactionId) ABContactAPI. updateContact(xml)
See also <ul style="list-style-type: none">• “Setting Guidewire Transaction IDs” on page 75 in the <i>Integration Guide</i>• “ABCClientAPI Interface” on page 240 in the <i>Contact Management Guide</i>• “ABContactAPI Web Service” on page 233 in the <i>Contact Management Guide</i>	
IReviewSummaryAPI.gs An RPC-E web service available to core applications for creating and deleting review summaries corresponding to vendor service provider reviews in ClaimCenter. Package name – gw.webservice.ab.ab700.reviewsummary	ABVendorEvaluationAPI.gs A WS-I compliant web service for ClaimCenter use to send and receive information about vendor provider service reviews with ContactManager. Package name – gw.webservice.ab.ab800.abvendorevaluationapi To find and open the class in Studio, use the Class Search dialog in non-project mode. Note: Press CTRL+N twice to turn on non-project class searching. See “ABVendorEvaluationAPI Web Service” on page 237 in the <i>Contact Management Guide</i>
ContactIntegrationXMLMapper.gs Name of the class that maps contact data as XML between ContactManager and the core applications. Package name – gw.webservice.ab.ab700.abcontactapi	ContactMapper This XML mapping class has been completely changed in ContactManager 8.0. Package name – gw.contactmapper.ab800 See “ContactMapper Class” on page 243 in the <i>Contact Management Guide</i> .

ContactManager 7.0	ContactManager 8.0
	ContactIntegrationXMLMapper.gs This XML mapping class supports integration with version 7.0 core applications.
	Package name – <code>gw.contactmapper.ab700</code>
ClientSystemPlugin Name of the plugin interface: <code>ClientSystemPlugin.java</code> . Package name: <code>gw.plugin</code> .	ClientSystemPlugin No name change. Read-only file is now visible in Studio if you do a CTRL+N search for <code>ClientSystemPlugin</code> .
ClaimSystemPlugin.xml Name of plugin registry. Navigate in Resource pane to configuration → Plugins → gw → plugin → <code>ClientSystemPlugin</code>	ClaimSystemPlugin.gwp Name of plugin registry. Navigate in Project window to configuration → config → Plugins → Registry
CCClaimSystemPlugin Plugin class to register when ClaimCenter 7.0 is installed. Extends <code>AbstractClientSystemPlugin.gs</code> . Package name – <code>gw.plugin.claim.cc700</code> .	CCClaimSystemPlugin (cc800 version) Plugin class to register when ClaimCenter 8.0 is installed. Extends <code>ClientSystemPlugin800.gs</code> . Package name – <code>gw.plugin.claim.cc800</code> . When ClaimCenter 8.0 is installed with ContactManager 8.0, register this plugin implementation.
	CCClaimSystemPlugin (cc700 version) Plugin class to register when ClaimCenter 7.0 is installed. Extends <code>ClientSystemPlugin700.gs</code> . Package name – <code>gw.plugin.claim.cc700</code> . When ClaimCenter 7.0 is installed with ContactManager 8.0, register this plugin implementation.
PolicySystemPlugin.xml Name of plugin registry. Navigate in Resource pane to configuration → Plugins → gw → plugin → <code>ClientSystemPlugin</code>	PolicySystemPlugin.gwp Name of plugin registry. Navigate in Project window to configuration → config → Plugins → Registry
PCPolicySystemPlugin Plugin class that extends <code>AbstractClientSystemPlugin.gs</code> . Package name – <code>gw.plugin.policy.pc700</code> .	PCPolicySystemPlugin (pc800 version) Plugin class that extends <code>ClientSystemPlugin800.gs</code> . Package name – <code>gw.plugin.policy.pc800</code> . When PolicyCenter 8.0 is installed with ContactManager 8.0, register this plugin implementation.
	PCPolicySystemPlugin (pc700 version) Plugin class that extends <code>ClientSystemPlugin700.gs</code> . Package name – <code>gw.plugin.policy.pc700</code> . When PolicyCenter 7.0 is installed with ContactManager 8.0, register this plugin implementation.
BillingSystemPlugin.xml Name of plugin registry. Navigate in Resource pane to configuration → Plugins → gw → plugin → <code>ClientSystemPlugin</code>	BillingSystemPlugin.gwp Name of plugin registry. Navigate in Project window to configuration → config → Plugins → Registry

ContactManager 7.0	ContactManager 8.0
BCBillingSystemPlugin Plugin class that extends <code>AbstractClientSystemPlugin.gs</code> . Package name – <code>gw.plugin.policy.bc700</code>	BCBillingSystemPlugin (bc800 version) Plugin class that extends <code>ClientSystemPlugin800.gs</code> . Package name – <code>gw.plugin.billing.bc800</code> . When BillingCenter 8.0 is installed with ContactManager 8.0, register this plugin implementation.
ABCClientAPI.gs The interface implemented by core applications to provide a way for ContactManager to call into those applications. Package name – <code>gw.webservice.ab.ab700abcontactapi</code> .	ABCClientAPI.gs (ab800 version) This interface has been updated to support pending contact changes. Package name – <code>gw.webservice.contactapi.ab800</code> To find and open the class in Studio, use the Class Search dialog in non-project mode. Note Press CTRL+N twice to turn on non-project class searching. See “ABCClientAPI Interface” on page 240 in the <i>Contact Management Guide</i> .
	ABCClientAPI.gs (ab700 version) This interface supports ContactManager 7.0 and is in a new package. Package name – <code>gw.webservice.contactapi.ab700</code>

ABContactAPI

This web service has some new methods and some changes to method parameters. You must fetch updates in Guidewire Studio for your core application for the ContactManager web service `ABContactAPI`. In addition, if you have customized or extended any of these methods, you will need to port your code to the new class.

The changes are as follows:

- Pre-existing methods no longer use the `transactionID` parameter. To set this parameter, you now call a separate method, `ContactAPIUtil.setTransactionId(ABContactAPI.Config, transactionId)`. See also “Setting Guidewire Transaction IDs” on page 80 in the Integration Guide.

- The following methods are new:

Method	Parameters	Description
createContactPendingApproval	abContactXML – Contact information in XmlBackedInstance format. updateContext – User, entity, and application information sent by core application.	Creates a new contact of the type specified and sets its status to APPROVAL_NEEDED. Returns an AddressBookUIDContainer containing IDs for the Contact and child objects and the update context and transaction ID. This method is called by the core application because the core application user creating the contact does not have permission to create a contact. Contact information is expected to be in XmlBackedInstance format. Calls ValidateABContactCreationPlugin to ensure that there is enough data to create the contact. If not, the method returns RequiredFieldException to the calling application. If there is enough data and no other exceptions are thrown, the method creates a new ABContact of the subtype specified by abContactXML. The method then populates the new entity with data it retrieves by calling ContactIntegrationMapper.populateABContactFromXML and sets its status to APPROVAL_NEEDED.
getSpecialistServices	contactLinkID – LinkID of the contact that has specialist services	Gets the specialist services associated with the contact passed in the parameter. Returns an array of ABContactAPISpecialistService objects, or null if the contact has no specialist services.
updateContactPendingApproval	abContactXML – Contact information in XmlBackedInstance format. updateContext – User, entity, and application information sent by core application.	Submits for approval an update to an existing contact that is pending until approved. The core application calling this method has determined that the user updating the contact does not have permission to do so. Contact information is expected to be in XmlBackedInstance format. If no existing ABContact can be found based on the abContactXML.LinkID, the method throws BadIdentifierException. If an ABContact entity is found with this LinkID, this method creates a PendingUpdate entity for the contact.

ABClientAPI

This interface has some new methods and some changes to method parameters. If you have customized the core application class that implements this interface, you must implement the new methods of this interface. You can use the implementation class in the core application as an example.

The changes are as follows:

- Pre-existing methods no longer use the `transactionID` parameter. To set this parameter, you now call a separate method, `ContactAPIUtil.setTransactionId(ABContactAPI.Config, transactionId)`. See also “Setting Guidewire Transaction IDs” on page 80 in the Integration Guide.

- The following methods are new:

Method	Parameters	Description
pendingCreateApproved	context – An ABCClientAPIPending ChangeContext object providing information on the user requesting this change.	Notifies the client system that a pending contact creation it submitted has been approved by ContactManager. The client application can use the information in the context parameter to inform the user who submitted the request that the contact was created. Additionally, the core application can update the sync status of the contact and post an appropriate message.
pendingUpdateApproved	context – An ABCClientAPIPending ChangeContext object providing information on the user requesting this change.	Notifies the client system that a pending update it submitted has been approved by ContactManager. The client application can use the information in the context parameter to inform the user who submitted the change that the change was approved. Additionally, the core application can update the sync status of the contact and post an appropriate message.
pendingCreateRejected	context – An ABCClientAPIPending ChangeContext object providing information on the user requesting this change.	Notifies the client system that a pending contact creation it submitted has been rejected by ContactManager. The client application can use the information in the context parameter to inform the user who submitted the contact to be created that the creation was rejected. Additionally, the core application can update the sync status of the contact and post an appropriate message.
pendingUpdateRejected	context – An ABCClientAPIPending ChangeContext object providing information on the user requesting this change.	Notifies the client system that a pending update it submitted has been rejected by ContactManager. The client application can use the information in the context parameter to inform the user who submitted the change that the change was rejected. Additionally, the core application can update the sync status of the contact and post an appropriate message.

For more information on this interface and the core application classes that implement, see:

- “ABCClientAPI Interface” on page 240 in the *Contact Management Guide*.

Changes to Contact Mapping in ContactManager

As described in “Configuration File Changes in ContactManager” on page 194, the ContactManager 7.0 contact XML mapping file `ContactIntegrationXMLMapper` is now called `ContactMapper`. If you have customized `ContactIntegrationXMLMapper`, you will need to port your code to `ContactMapper`, which has been completely refactored to provide simpler code for adding foreign keys and array references.

For a description of this class, see “ContactMapper Class” on page 243 in the *Contact Management Guide*.

Changes to Contact Search Functionality in ContactManager

The changes in search web service names are listed in the table at “Configuration File Changes in ContactManager” on page 194. In addition to those file name changes, there are other changes that affect how searches are performed across the core applications and ContactManager.

Additionally, to see how to use the search classes in your own extensions, see “Searching for Contacts” on page 83 in the *Contact Management Guide*.

Changes to Typekey Criteria in Contact Searches

ABContactAPI now exposes typekeys used in Contact search as strings.

In the previous release, ContactManager 7.0 exposed these typekeys as enum values that ClaimCenter could access from `ContactSearchMapper` as follows:

```
searchCriteriaInfo.VendorType =  
    wsi.remote.gw.webservice.ab.ab700.abcontactapi.enums.VendorType.forGosuValue(  
        searchCriteria.VendorType.Code)
```

In ContactManager 8.0, ABContactAPI exposes typekeys as strings, such as the `VendorType` typelist:

```
@WsiExposeEnumAsString(typekey.VendorType)
```

Now that these typekeys are simple strings, the core application method call is also much simpler. For example, the ClaimCenter `ContactSearchMapper` code for `VendorType` search criterion is now:

```
searchCriteriaInfo.VendorType = searchCriteria.VendorType.Code
```

Upgrading from 3.0.x

This part describes how to perform an upgrade from BillingCenter 3.0.4 or newer to 8.0.4. You can upgrade the database to BillingCenter directly from 3.0.x versions that are 3.0.4 or newer.

If you are upgrading from a BillingCenter 3.0.x version that is prior to 3.0.4, you must first upgrade your BillingCenter database to the latest version of 3.0.x. Then you can upgrade your configuration and database to version 8.0.4. To do this, obtain the latest 3.0.x release, and upgrade your extensions, typelists, and metadata. Consult Guidewire Services for assistance.

If you are upgrading from BillingCenter 7.0.x, see “Upgrading from 7.0.x” on page 93 instead.

This part includes the following topics:

- “Upgrading the BillingCenter 3.0.x Configuration” on page 203
- “Upgrading the BillingCenter 3.0.x Database” on page 249
- “Upgrading Integrations and Gosu from 3.0.x” on page 321

Upgrading the BillingCenter 3.0.x Configuration

This topic describes how to upgrade the BillingCenter configuration from version 3.0.x.

If you are upgrading from an 8.0.x version, see “Upgrading the BillingCenter 8.0.x Configuration” on page 29 instead.

If you are upgrading from a 7.0.x version, see “Upgrading the BillingCenter 7.0.x Configuration” on page 95 instead.

This topic includes:

- “Obtaining Configurations” on page 204
- “Creating a Configuration Backup” on page 208
- “Removing Patches” on page 208
- “Removing Language Packs” on page 208
- “Updating Infrastructure” on page 208
- “Upgrading the BillingCenter 3.0 Configuration to 7.0” on page 208
- “BillingCenter 7.0 Upgrade Tool Automated Steps” on page 209
- “Configuring the BillingCenter 8.0 Upgrade Tool” on page 211
- “Launching the BillingCenter 8.0 Configuration Upgrade Tool” on page 213
- “BillingCenter 8.0.4 Configuration Upgrade Tool Automated Steps” on page 214
- “Using the BillingCenter 8.0.4 Upgrade Tool Interface” on page 221
- “Configuration Merging Guidelines” on page 227
- “Data Model Merging Guidelines” on page 228
- “Preserving Payment Method Details” on page 233
- “Migrating BCContact and PaymentDetails Extensions” on page 235

- “Changes to the Logging API” on page 235
- “Changes to Iterators in PCF Files” on page 238
- “Updating Namespace on Files Loaded by GX Models” on page 239
- “Adding DDL Configuration Options to database-config.xml” on page 239
- “Merging Changes to Field Validators” on page 239
- “Renaming PCF files According to Their Modes” on page 240
- “Updating Rounding Mode Parameter” on page 240
- “Merging compatibility-xsd.xml” on page 240
- “Merging Display Properties” on page 241
- “Merging Other Files” on page 242
- “Migrating to 64-bit IDs During Upgrade (SQL Server Only)” on page 242
- “Fixing Gosu Issues” on page 243
- “Upgrading Rules to BillingCenter 8.0.4” on page 245
- “Running PCF Iterator Upgrade” on page 246
- “Translating New Display Properties and Typecodes” on page 246
- “Validating the BillingCenter 8.0.4 Configuration” on page 247
- “Building and Deploying BillingCenter 8.0.4” on page 248

Obtaining Configurations

Configuration refers to everything related to the application except the database. This includes configuration files such as typelists and PCF files, the file structure, web resources, Gosu (formerly GScript) classes, rules, plugins, libraries, localization files, and application server files.

The upgrade process involves multiple configurations. This guide refers to these configurations as base, customer, intermediate, and target.

Base – The unedited, original configuration on which you based your customer configuration. The base configuration in BillingCenter 3.0 is included in directories within /modules other than /configuration.

Customer – The configuration you are now using and will upgrade. This is the base configuration of the BillingCenter version that you currently run with your custom configuration applied.

Intermediate – A BillingCenter 7.0 installation. You run the automated steps of the BillingCenter 7.0 Configuration Upgrade Tool to convert your BillingCenter 3.0 customer configuration to BillingCenter 7.0. You do not need to perform a manual configuration merge to BillingCenter 7.0. Then you use the BillingCenter 8.0.4 Configuration Upgrade Tool to convert the BillingCenter 7.0 configuration to BillingCenter 8.0.4. You do not need to upgrade the database to 7.0 before upgrading it to 8.0.4. You can upgrade the database from 3.0 to 8.0.4 in one procedure.

Target – The unedited, original configuration of BillingCenter 8.0.4 on which your upgraded configuration will be based. Do not make any modifications to the target configuration prior to completing the configuration upgrade. Do not start Guidewire Studio for the target configuration until you have completed the configuration upgrade.

Guidewire grants you access to an `ftp` site from which you download the intermediate BillingCenter 7.0 configuration and the target BillingCenter 8.0.4 configuration.

Unzip the target BillingCenter 8.0.4 and intermediate BillingCenter 7.0 into separate directories.

IMPORTANT Set all files in your custom configuration, and the intermediate and target configurations to writable before beginning the upgrade.

Viewing Differences Between Base and Target Releases

To view an inventory of the differences between the base release and the target release, download and carefully review the *Upgrade Diffs Report* from the Guidewire Resource Portal.

1. Open a browser to <https://guidewire.hivelive.com/pages/home>.
2. Click Project Center → Upgrade Services.
3. Click Review the Upgrade Diff Reports.
4. Click BillingCenter.
5. Click Upgrade Diff Reports - BillingCenter or Upgrade Diff Reports - ContactManager.
6. Click Upgrade From *base version*.
7. Click Upgrade To 8.0.4.

Specifying Configuration Locations for BillingCenter 7.0 Upgrade Tool

The BillingCenter 7.0 Configuration Upgrade Tool requires the location of the custom BillingCenter environment that you are upgrading. Define this path in the `BillingCenter/modules/ant/upgrade.properties` file of the intermediate BillingCenter 7.0 environment. Remove the pound sign, '#', preceding each property to uncomment the line and then specify values. Use double backslashes in paths. For example, `C:\\BillingCenter`. You do not need to use quotes for paths that include spaces.

The following properties are configured in `upgrade.properties`.

Property	Description
<code>upgrader.priorversion.dir</code>	Path to the top-level BillingCenter directory of the current customer environment. This directory contains <code>/bin</code> and other BillingCenter directories.
<code>upgrader.editor.tool</code>	Path to an executable editing tool. You do not need to use a text editor for the intermediate upgrade. However, you can point to one if you want to examine files following the automated steps of the intermediate upgrade.
<code>upgrader.diff.tool</code>	Path to an executable difference editor tool, such as Araxis Merge Professional or Beyond Compare 3 Professional, also known as a merge tool, used for two-way merges. If your difference editor supports both two and three-way merges, you can use the same value for <code>upgrader.diff.tool</code> and <code>upgrader.merge.tool</code> . You do not need to use a difference editor tool for the intermediate upgrade. However, you can point to one if you want to examine files following the automated steps of the intermediate upgrade.
<code>upgrader.merge.tool</code>	Path to an executable difference editor tool, such as Araxis Merge Professional or Beyond Compare 3 Professional, also known as a merge tool, used for three-way merges. You do not need to use a merge tool for the intermediate upgrade. However, you can point to one if you want to examine files following the automated steps of the intermediate upgrade. If your difference editor supports both two and three-way merges, you can use the same value for <code>upgrader.diff.tool</code> and <code>upgrader.merge.tool</code> . You might need to configure the display of your merge tool to show three panels.

Property	Description
<code>upgrader.merge.tool.arg.order</code>	<p>You do not need to use the merge tool during upgrade to the intermediate configuration, so you can ignore this property during the upgrade.</p>
	<p>This property defines the display order, from left to right, for versions of a file viewed in the difference editor tool. By default, the tool displays, left to right, the versions of a file in this order:</p> <p><code>NewBase PriorBase PriorCustom</code></p> <p>in which:</p>
	<p><code>NewBase</code> is the unmodified intermediate version provided with BillingCenter 7.0.</p>
	<p><code>PriorBase</code> is the original base version.</p>
	<p><code>PriorCustom</code> is your configured version.</p>
	<p>The order of these values controls the display order in the difference editor tool. If the tool displays just two versions, it uses the same relative order.</p>
	<p>By default, the display order places the old base version of a file in the center. The original base version is the common ground between the new uncustomized version and the old customized version. Guidewire changed the old base version to create the new target version, and you changed the old base version to create the customized version in your configuration. With the old base version in the center, you can incorporate both sets of changes to create a customized target version.</p>
	<p>The default order enables you to merge changes from the left and right to the center and save the merged version. If you use another difference editor, you might need a different order to achieve the same result. Samples are shown below for various difference engines:</p>
	<p>Araxis Merge Professional</p>
	<pre>upgrader.merge.tool.arg.order = NewFile OldFile ConfigFile</pre>
	<p>Beyond Compare 3 Professional</p>
	<pre>upgrader.merge.tool.arg.order = NewFile ConfigFile Oldfile</pre>
	<p>P4Merge</p>
	<pre>upgrader.merge.tool.arg.order = OldFile NewFile ConfigFile</pre>
	<p>You might need to configure the display of your merge tool to show three panels.</p>
<code>upgrader.steps.class</code>	<p>The class to run to execute the configuration upgrade automated steps. Leave this property commented out.</p>
<code>exclude.pattern</code>	<p>A regular expression pattern for paths of files for the Configuration Upgrade Tool to mark as unmergeable. Typically, you use <code>exclude.pattern</code> to specify source control metadata files. Samples are provided in <code>upgrade.properties</code> for CVS and SVN.</p>

Creating a Configuration Backup

Prepare the environment so that you can make a total recovery of the original installation if you run into problems during the upgrade.

Guidewire recommends that you track BillingCenter configuration changes in a source code control system. Before upgrading, have a labeled version of your entire pre-upgrade BillingCenter configuration folder (`modules/configuration`) from your custom configuration. A labeled version is a named collection of file revisions.

As an even stronger precaution, make a backup of the same installation directories.

Removing Patches

If you have applied any patches from Guidewire to BillingCenter, remove the patches before you run the configuration upgrade. Patches are specific to the pre-upgrade version. If you do not remove the JAR files for patches, the Configuration Upgrade Tool copies the JAR files to the upgraded configuration. In that case, your upgraded configuration will be using a JAR file that is not current. Patches are typically installed by adding JAR files within `modules/configuration/deploy`.

Removing Language Packs

If you have language packs installed, you must remove the language packs before upgrading BillingCenter. See “Upgrading Display Languages” on page 30 in the *Globalization Guide*.

Updating Infrastructure

Before starting the upgrade, have the supported server operating systems, application server and database software, JDK, and client operating systems for the target version. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

For SQL Server, after you upgrade the database server software, run the following command to set the compatibility level:

```
ALTER DATABASE databaseName SET COMPATIBILITY_LEVEL = 110
```

Upgrading the BillingCenter 3.0 Configuration to 7.0

Use the BillingCenter 7.0 Configuration Upgrade Tool to upgrade the configuration from 3.0 to 7.0. You do not need to perform a merge at this step. You only need to run the automated steps of the BillingCenter 7.0 Configuration Upgrade Tool.

Launching the BillingCenter 7.0 Configuration Upgrade Tool

To launch the BillingCenter 7.0 Configuration Upgrade Tool

1. Open a command window.
2. Navigate to the `modules/ant` directory of the BillingCenter 7.0 configuration.

3. Execute the following command:

```
ant -f upgrade.xml upgrade > upgrade_log.txt
```

You can specify any file to log messages and exceptions. For a typical installation, this command takes a few hours. If it can not complete, restart it.

The Configuration Upgrade Tool performs a number of automated steps. Once the tool completes the automated steps, it opens a user interface. The interface opens whether the automated steps were successful or not.

4. Review the log file or console to verify the automated steps were successful.

5. Close the BillingCenter 7.0 Configuration Upgrade Tool interface. You do not need to use the BillingCenter 7.0 Configuration Upgrade Tool interface. You only need to run the tool for the automated steps it performs.

Restarting the Configuration Upgrade Tool

To restart the upgrade, first use the `clean` command to empty the working directories.

```
ant -f upgrade.xml clean
```

BillingCenter 7.0 Upgrade Tool Automated Steps

The Configuration Upgrade Tool performs a number of automated steps. Review these steps as some might require manual intervention if there is an issue.

Moving Typelist Localizations into typelist.properties Files

This upgrade step refactors typelist localizations. In versions of BillingCenter prior to 7.0, typecode localizations were stored in typelist XML files. This required you to modify and possibly extend a typelist definition to localize typecode names and descriptions.

In BillingCenter 7.0, typecode localizations are stored in `typelist.properties` files that follow the same pattern as display keys. These files are stored per module per locale as:

```
module/config/locale/locallename/typelist.properties
```

The upgrade functions on a per-module basis and performs the following actions:

1. Scans the `config/metadata` and `config/extension` directories for all typelist files.
2. Opens each file and parses the XML.
3. Removes all typecode localizations, caching them in memory keyed by locale.
4. Saves the edited XML back to the typelist file.
5. After all typelist files are scanned, the upgrader iterates over the cached localizations.
6. For each locale, creates the `typelist.properties` file and writes out the localizations.

Removing Redundant TTX Files

Many TTX files in the `configuration` module exist only because a typelist was localized. That leaves TTX files that contain no real customizations in the `configuration` module.

As of BillingCenter 7.0, all typelist localizations are stored in `typelist.properties` files. The prior upgrade step moves the typelist localizations to `typelist.properties`.

This upgrade step deletes TTX files that were created in prior versions that only contain localization information.

Removing searchTypeVisible Attribute from DateCriterionChoiceInputNode

The configuration upgrade updates PCF files to remove the `searchTypeVisible` attribute from `DateCriterionChoiceInput` elements.

Copying Display Properties Files into Target Configuration

This step copies `display.properties` files from the `locale` directories of the working configuration module to the target configuration module.

If the file already exists in the target configuration, the tool skips the copy and logs a message. This is a precaution to make sure that the Configuration Upgrade Tool does not overwrite customized `display.properties` files if you run the tool again.

Copying Custom Rules and Adding BillingCenter 8.0.4 Default Rules

This step copies customized rules to the target configuration `modules/configuration/config/rules/rules` directory.

This step also copies the default rules provided with BillingCenter 8.0.4 to a BillingCenter 8.0.4 folder within the `modules/configuration/config/rules/rules` directory of the target configuration. This is so you have a copy of the default rules in a folder in Studio that you can use to compare with your custom rules.

See “Upgrading Rules to BillingCenter 8.0.4” on page 245.

Referencing XSD Files

Guidewire now provides a `compatibility-xsd.xml` file in `modules/configuration/config/registry`. This file contains a list of the XSD files that exist in BillingCenter. This upgrade step locates XSD files and places an entry in `compatibility-xsd.xml` for each XSD file.

Removing AdminTable Delegate from Custom Extensions

This step removes the `AdminTable` delegate from custom extensions.

Converting sessiontimeoutsecs Security Element to Parameter

This step converts the `security` element `sessiontimeoutsecs` in `config.xml` to a parameter. For example, the tool converts:

```
<security sessiontimeoutsecs="10800"/>  
to  
<param name="SessionTimeoutSecs" value="10800"/>
```

The value is preserved during the conversion.

Removing Redundant Batch Server Parameter

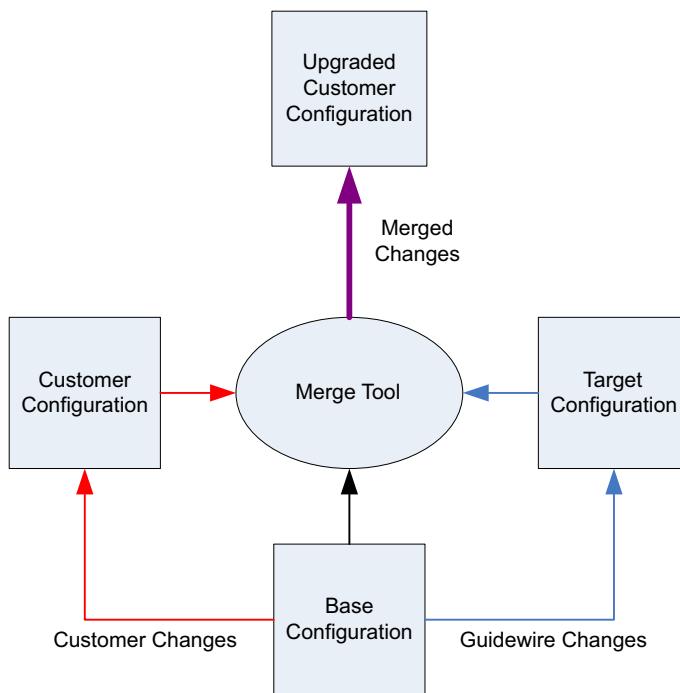
This step removes the redundant `BatchServer` parameter from `config.xml`.

Configuring the BillingCenter 8.0 Upgrade Tool

After you have run the automated steps of the BillingCenter 7.0 Configuration Upgrade Tool, run the BillingCenter 8.0.4 Configuration Upgrade Tool to complete the upgrade from 7.0 to 8.0.4. First, specify locations of configurations and tools used to merge the configurations. Then run the BillingCenter 8.0.4 Configuration Upgrade Tool.

To upgrade your configuration, merge Guidewire changes to the base configuration with your changes. The Configuration Upgrade Tool, provided by Guidewire with the target configuration, facilitates this process.

The following figure shows how you use these configurations to create a merged configuration. The merged configuration combines your changes to the original base configuration (the customer configuration) and Guidewire changes to the base configuration (the target configuration). The original base configuration provides a basis for comparison.



The BillingCenter 8.0.4 Configuration Upgrade Tool depends on the following tools:

- **Text Editor** – An ASCII text editor you use to edit programs and similar files, for example, Notepad, WordPad or Textpad. This editor must not put additional characters in files, as Word does.
- **Merge Tool** – An editor which displays two or three versions of a file, highlights the differences between them, and allows you to perform basic editing functions on them. Also known as a “diff tool.” Examples include Araxis Merge Professional and Beyond Compare Professional. If using Beyond Compare Professional, see “Considerations for Using Beyond Compare Professional” on page 213. Configure the merge tool to ignore end of line characters to reduce the number of potential false positives during the configuration upgrade step.

IMPORTANT The merge tool that you use must support three-way file comparison and merging. During the configuration upgrade, for some files you will need to compare three versions: the original base version, the new version and your customized version.

The Configuration Upgrade Tool needs the location of the BillingCenter environment that you will upgrade. The tool stores all versions of files to be merged and merge results in a `tmp` directory that it creates within the target environment. Define paths to the configuration and tools in the `BillingCenter/modules/ant/upgrade.properties` file of the target BillingCenter 8.0.4 environment. Remove the pound sign, '#', preceding each property to uncomment the line and then specify values. Use double backslashes in paths. For example, `C:\\\\BillingCenter`. You do not need to use quotes for paths that include spaces.

The following properties are configurable in `upgrade.properties`.

Property	Description
<code>upgrader.priorversion.dir</code>	Path to the top-level BillingCenter directory of the current customer environment. This directory contains <code>/bin</code> and other BillingCenter directories.
<code>upgrader.editor.tool</code>	Path to an executable editing tool.
<code>upgrader.diff.tool</code>	Path to an executable difference editor tool, such as Araxis Merge Professional or Beyond Compare Professional, also known as a merge tool, used for two-way merges. If your difference editor supports both two and three-way merges, you can use the same value for <code>upgrader.diff.tool</code> and <code>upgrader.merge.tool</code> .
<code>upgrader.merge.tool</code>	Path to an executable difference editor tool, such as Araxis Merge Professional or Beyond Compare Professional, also known as a merge tool, used for three-way merges. The merge tool specified for <code>upgrader.merge.tool</code> must support three-way file comparison and merging. If your difference editor supports both two and three-way merges, you can use the same value for <code>upgrader.diff.tool</code> and <code>upgrader.merge.tool</code> . You might need to configure the display of your merge tool to show three panels.
<code>upgrader.merge.tool.arg.order</code>	The order of command line arguments to the difference editor tool specified by <code>upgrader.merge.tool</code> . The command line arguments available to the difference editor typically include the display order, from left to right, for versions of a file viewed in the difference editor. The available options are: NewBase is the unmodified target version provided with BillingCenter 8.0.4. PriorBase is the original base version. PriorCustom is your configured version. Resulting is the merged output file. The order of these values controls the display order in the difference editor tool. If the tool displays just two versions, it uses the same relative order. By default, the display order places the old base version of a file in the center. The old base version is the common ground between the new uncustomized version and the old customized version. Guidewire changed the old base version to create the new target version, and you changed the old base version to create the customized version in your configuration. With the old base version in the center, you can incorporate both sets of changes to create a customized target version. The default order enables you to merge changes from the left and right to the center and save the merged version. If you use another difference editor, you might need a different order to achieve the same result. Samples are shown below for various difference engines: Araxis Merge Professional <code>upgrader.merge.tool.arg.order = NewBase PriorBase PriorCustom</code> Beyond Compare Professional <code>upgrader.merge.tool.arg.order = NewBase PriorCustom PriorBase Resulting</code> P4Merge <code>upgrader.merge.tool.arg.order = PriorBase NewBase PriorCustom</code> You might need to configure the display of your merge tool to show three panels.

Property	Description
upgrader.steps.class	The class to run to execute the configuration upgrade automated steps. If you are upgrading BillingCenter 3.0 or newer, then leave this property commented out.
exclude.pattern	A regular expression pattern for paths of files for the Configuration Upgrade Tool to mark as unmergeable. Typically, you use exclude.pattern to specify source control metadata files. Samples are provided in upgrade.properties for CVS and SVN.

Considerations for Using Beyond Compare Professional

By including the Resulting parameter on the upgrader.merge.tool.arg.order property for Beyond Compare, you can avoid an extra manual step to specify the output file on each merge.

To configure the Configuration Upgrade Tool for Beyond Compare Professional

1. Create a new file in a text editor.
2. Add the following text to the new file, modifying the location of the Beyond Compare Professional executable if necessary.


```
@echo off
"C:\Program Files (x86)\Beyond Compare 3\BCompare.exe" %1 %2 %3 %4
```
3. Save the file as merge.cmd in the BillingCenter/modules/ant directory.
4. Open the BillingCenter/modules/ant/upgrade.properties file of the target BillingCenter 8.0.4 environment.
5. Set the upgrader.diff.tool property to the location of the Beyond Compare Professional executable. For example:


```
upgrader.diff.tool = "C:\\Program Files (x86)\\Beyond Compare 3\\BCompare.exe"
```
6. Set the upgrader.merge.tool property to a command line script file. For example:


```
upgrader.merge.tool = ".\\modules\\ant\\merge.cmd"
```
7. Set the upgrader.merge.tool.arg.order property:


```
upgrader.merge.tool.arg.order = NewBase PriorCustom PriorBase Resulting
```
8. Set the remaining properties in upgrade.properties as described in the table.
9. Save upgrade.properties.

Launching the BillingCenter 8.0 Configuration Upgrade Tool

To launch the BillingCenter 8.0 Configuration Upgrade Tool

1. Open a command window.
2. Navigate to the modules/ant directory of the target configuration.
3. Execute the following command:

```
ant -f upgrade.xml upgrade > upgrade_log.txt
```

You can specify any file to log messages and exceptions.

The Configuration Upgrade Tool first copies the modules of the base environment to a tmp/cfg-upgrade/modules directory in the target environment. The base environment is specified by the upgrader.priorversion.dir property in ant/modules/upgrade.properties in the target environment.

The Configuration Upgrade Tool then performs a number of automated steps, described later in this topic. Once the tool completes the automated steps, it opens a user interface. The interface opens whether the automated steps were successful or not. Review the log file or console before proceeding with the manual merge process.

Note: The Configuration Upgrade Tool does not upgrade rules. Merge rules after completing the rest of the configuration upgrade.

Restarting the Configuration Upgrade Tool

The Configuration Upgrade Tool stores work in progress by recording which files you have marked resolved in the `accepted_files.lst` file. This file is stored in the `merge` folder of the target environment. You can close the interface and restart it later without losing your work in progress.

If you do want to start the upgrade over, use the `clean` command to empty the working directories.

```
ant -f upgrade.xml clean
```

WARNING If you empty the `tmp` directory after beginning to merge, you lose all completed merges that you have not resolved and moved into the target configuration directory.

BillingCenter 8.0.4 Configuration Upgrade Tool Automated Steps

The Configuration Upgrade Tool prepares for the manual configuration merge process by performing a number of automated steps. Review these steps before proceeding with the configuration merge. Understanding these automated steps helps to understand some file changes you will see when merging the configuration. Finally, some steps might require manual intervention if there is an issue.

Removing Template Pages

The Configuration Upgrade Tool deletes PCF template pages. These pages have a `<TemplatePage>` root element. The upgrade also removes `<EntryPoint>` elements that reference template pages. Template pages have been replaced by SOAP-based data integration in BillingCenter 8.0. See “Template Page PCF Files Removed” on page 42 in the *New and Changed Guide*.

Updating PCF Files

The Configuration Upgrade Tool performs the following modifications to PCF files:

- Removes the `reflectOnBottom` attribute. This attribute was used to display the a virtual toolbar at the bottom of a page. The attribute was removed because the user interface needs to match the server configuration. No alternative configuration is available.
- Converts all `postOnChange` attributes on a value widget to a child `PostOnChange` node. For example, the upgrade converts:

```
<Input id="xxx" postOnChange="true" onChange="someMethod()" disablePostOnEnter="doEvaluation()"/>  
to:
```

```
<Input id="xxx">  
  <PostOnChange onChange="someMethod()" disablePostOnEnter="doEvaluation()"/>  
</Input>
```

- Removes the `showNoneSelected` attribute from all `DetailView` inputs that are bound to a value. Setting `showNoneSelected=false` would suppress the **None Selected** option from drop-down lists and would default to the first option. This type of configuration was incorrect because the selection of the option was generally programmatically incorrect and was often used as a shortcut instead of specifying an explicit default. Verify all removals to ensure there is not any dependent logic. If there is, specify an explicit default in the page configuration.

- Removes the `showNoneSelected` attribute from all `<ValueCellType>` nodes. See the above note about removal of the `showNoneSelected` attribute from all `DetailView` inputs that are bound to a value.
- Removes the `numDataEntriesPerRow` and `transposed` attributes from `RowIteratorNode` elements. Transposed lists are a relatively rare configuration. If you had one in your configuration, use a traditional list view.
- Removes `<DetailViewPanel>` elements from `<ButtonCell>`, `<ButtonInput>`, and `<ToolbarButtonType>` elements. Detail views can no longer be embedded inside buttons.
- Converts `valueWidth` attributes on cell widgets to `value` attributes. As of 8.0, BillingCenter sizes cells by heuristics rather than content, so `valueWidth` is no longer necessary.
- If all cells in a row have the `useHeaderStyle="true"` property, the upgrade moves the property to the row level. A list can only have one header. See below.
- Updates rows to rename the `useHeaderStyle` property to `renderAsSmartHeader`. The property is renamed because the header functionality is more than styling. When a row is rendered as a smart header, all the row header interactive features are made available.
- Renames `<ContentCell>` elements to `<Link>`.
- Converts `<Cell>` elements within `<ColumnFooter>` to `<TextCell>` elements.
- Removes any element that is not a `<TextCell>` element from `<ColumnFooter>` elements.
- Removes `<ColumnHeader>` elements from `<CellType>` elements.
- Remove `<DetailViewPanel>` from `<ContentCell>`. The upgrade performs the following steps. After the automatic upgrade, review your `<ContentCell>` configurations to manually verify the configuration and make any changes. Content cells cannot have editable detail views embedded in them. Review all removals to ensure functionality. If editable content is needed within a row of data, the recommended configuration is a list detail panel.
 - For any `<ContentCell>` that contains a `<DetailViewPanel>`, the upgrade renames the `<ContentCell>` to `<FormatCell>`.
 - For other types of `<ContentCell>`, the upgrade renames the element to `<LinkCell>`.
 - Removes elements that are not allowed in the `<FormatCell>`, such as `<DetailViewPanel>` and `<InputColumn>`. This strips out unnecessary container elements. No content will be removed.
 - Renames inputs in the `<DetailViewPanel>` to `<TextInput>` unless they are `<ContentInput>`, `<TextInput>`, or `<NoteBodyInput>`.
 - Removes attributes that were allowed on specific input elements but not on `<TextInput>`.
- Removes the `useHeaderStyle` attribute from all cells that can be bound to a value. The header style in 8.0 is a lot more extensive. Smart header capabilities have been added, in addition to the styling. Header capabilities are at the row level as opposed to the cell. If you are interested in highlighting content, there are a few other ways to achieve that. Review the PCF reference for a full list of attributes for that particular cell variant.
- Removes the `compress` attribute from `<DetailViewPanel>`.
- Removes the `compress` attribute from `<ListViewPanel>`.
- Removes the `compressIfSingleChild` attribute from `<InputGroup>`.
- Comments out `<ProgressCell>` elements. This was an uncommon widget that Guidewire has removed. If you were using it on some page and would like to continue to do so, create a list detail panel, and use the `ProgressInput` in the detail section instead.
- Removes the `refreshOnProgressComplete` attribute from `<ListViewPanel>` and `<Row>` elements. This is part of the removal of the `<ProgressCell>` widget.
- Removes the following attributes from `<ChartPanel>`:
 - `bgColor`
 - `border`
 - `displayPlotOutline`
 - `orientation`

- `sameSeriesColor`
- `threeD`
- `tooltip`

Guidewire cleaned up the `<ChartPanel>` schema as a part of simplification and a move to a more interactive experience.

- Removes the following attributes from `<DomainAxis>`:
 - `autoRange`
 - `autoRangeIncludesZero`
 - `tickUnit`
 - `upperMargin`
- Removes the `<Interval>` element.
- Removes the following attributes from `<RangeAxis>`:
 - `autoRange`
 - `autoRangeIncludesZero`
 - `tickUnit`
 - `upperMargin`
- Removes the `percentComplete` attribute from `<DataSeries>`.
- Removes the following from `<DualAxisDataSeries>`:
 - `autoRangeIncludesZero`
 - `lowerMargin`
 - `tickUnit`
 - `tooltip`
 - `upperMargin`
- Removes the following chart types from the `<ChartType>` enumerator:
 - `Waterfall`
 - `Gantt`
- Renames the following chart types in the `<ChartType>` enumerator:
 - `Dial` → `Gauge`
 - `Polar` → `Radar`
 - `Ring` → `Pie`
 - `StackedArea` → `Area` (there is no more distinction between a stacked vs non-stacked area)
 - `XYStep` → `XYLine`
 - `XYStepArea` → `XYArea`

Upgrading Work Queue Configuration

The Configuration Upgrade Tool makes the following changes to `work-queue.xml`:

- removes obsolete `minpollinterval` attribute.
- removes obsolete `orphansFirst` attribute.
- removes obsolete `checkInAfterError` attribute
- adds `retryInterval=7200` (the upgrade sets the value to 0 if `checkInAfterError` was `true`, or to the current value of `progressinterval` if `checkInAfterError` was not `true`.)

For more information about changes to work queue configuration, see “Changes to Work Queue Configuration” on page 76 in the *New and Changed Guide*.

Upgrading Database Configuration

The Configuration Upgrade Tool moves the database configuration from `config.xml` to `database-config.xml` and converts it to the BillingCenter 8.0 format.

As of BillingCenter 8.0, Guidewire has made the following changes to the database configuration:

- The `<database>` element no longer contains subelements with the following syntax:
`<param name="name" value="value">`
- For Oracle, the `<tablespacemapping>` elements have been replaced with a single `<tablespaces>` element. The `<tablespaces>` element is contained in an `<ora-db-ddl>` parent element. The `<tablespaces>` element includes the attributes `admin`, `index`, `op`, `staging`, `typelist`, and `lob`. These attributes correspond to the logical tablespaces defined in BillingCenter. You can use these attributes to map tablespaces that you have created to the logical tablespaces.
- For SQL Server, the `<tablespacemapping>` elements have been replaced with a single `<mssql-filegroups>` element. The `<mssql-filegroups>` element is contained in an `<mssql-db-ddl>` parent element. The `<mssql-filegroups>` element includes the attributes `admin`, `index`, `op`, `staging`, `typelist`, and `lob`. These attributes correspond to the logical tablespaces defined in BillingCenter. You can use these attributes to map file groups that you have created to the logical tablespaces.
- If a `<tablegroup>` element was contained in a `<database>` element that had an `env` attribute defined, the upgrade copies the `env` attribute onto the `<tablegroup>` element.
- If any of the following `<database>` attributes are defined, the upgrade copies them over to the `<database>` element in `database-config.xml`: `addforeignkeys`, `autoupgrade`, `checker`, `dbtype`, `env`, `name`, `printcommands`. The schema for these attributes has not changed.
- If any comments exist within the `<database>` element, the upgrade copies these comments to the `<database>` element in `database-config.xml`.
- If the `driver` attribute of the `<database>` element equals `dbcp`, the upgrade adds a `<dbcp-connection-pool>` element and copies the `jdbcUrl` parameter to the `jdbc-url` attribute of the `<dbcp-connection-pool>` element. If the original configuration did not include a `jdbcUrl` parameter, then the upgrade logs an error. If a `passwordFile` attribute is specified on the `<database>` element of the old configuration, the upgrade transfers the `passwordFile` attribute to the `<dbcp-connection-pool>` element. The upgrade converts any of the following parameters defined in the old configuration to attributes on the `<dbcp-connection-pool>` element:
 - `maxActive`
 - `maxIdle`
 - `maxWait`
 - `minEvictableIdleTimeMillis`
 - `numTestsPerEvictionRun`
 - `testOnBorrow`
 - `testOnReturn`
 - `testWhileIdle`
 - `timeBetweenEvictionRunsMillis`
 - `whenExhaustedAction`
- If the `driver` attribute of the `<database>` element equals `dbcp` and any of the following attributes are set, the upgrade creates a `<reset-tool-params>` element within the `<dbcp-connection-pool>` element:
 - `collation`
 - `oracle.tnsnames`
 - `system.username`
 - `system.password`

The upgrade then transfers any of these attributes that are defined to the new `<reset-tool-params>` element.

- If the `driver` attribute of the `<database>` element equals `jndi`, the upgrade adds a `<jndi-connection-pool>` element and copies the `jndi.datasource.name` parameter to the `datasource-name` attribute of the `<jndi-connection-pool>` element. If the original configuration did not include a `jndi.datasource.name` parameter, then the upgrade logs an error.
- If the old configuration includes an `<upgrade>` element within the `<database>` element, the upgrade adds an `<upgrade>` element to the `<database>` element of the new configuration.
- If the old configuration contains an `<upgrade>` element that includes an `oracleMarkColumnsUnused` attribute, the upgrade converts the attribute to a `deferDropColumns` attribute, preserving the value.
- If the old configuration contains an `<upgrade>` element that includes a `verifySchema` attribute, the upgrade copies this attribute to `<upgrade>` element of the new configuration.
- If the old configuration contains an `<upgrade>` element that contains an `<oracleddloptions>` or `<sqlserverddlopts>` element, the upgrade logs a warning. You must upgrade these elements manually.
- If the old configuration includes a `<databasestatistics>` element within the `<database>` element, the upgrade copies the `<databasestatistics>` element to the `<database>` element of the new configuration.
- For Oracle databases, if the `<database>` element includes any of the following parameters, the upgrade creates an `<oracle-settings>` element within the `<database>` element of the new configuration:
 - `queryRewriteEnabled`
 - `statisticsLevel`
 - `stored.outlines`
 - `UseDbResourceMgrCancelSql`

The upgrade converts any of the above parameters to attributes on the new `<oracle-settings>` element. The attributes have the following names:

- `query-rewrite`
- `statistics-level-all` (if any value is set for `statisticsLevel` in the old configuration, the upgrade sets the `statistics-level-all` attribute to `true` in the new configuration. The value `ALL` was the only valid value for the `statisticsLevel` parameter in the old configuration.)
- `stored-outline-category`
- `db-resource-mgr-cancel-sql`
- For SQL Server databases, if the `<database>` element includes either the `msjdbctracelevel` or `msjdbctracefile` parameter, the upgrade adds a `<sqlserver-settings>` element within the `<database>` element of the new configuration. The upgrade then converts the `msjdbctracelevel` and `msjdbctracefile` parameters to `jdbc-trace-level` and `jdbc-trace-file` attributes on the `<sqlserver-settings>` element.
- For SQL Server databases, if the `unicodecolumns` parameter is defined in the old configuration, the upgrade adds a `unicodecolumns` attribute to the `<sqlserver-settings>` element of the new configuration. If the `<sqlserver-settings>` element has not yet been created, the upgrade creates the element.
- If any `<tablespacemapping>` elements are defined in the old configuration, the upgrade creates an `<upgrade>` element within the `<database>` element of the new configuration if one does not yet exist. The upgrade then does the following, depending on the database type:
 - For Oracle, the upgrade adds an `<ora-db-ddl>` element within the `<upgrade>` element of the new configuration, if an `<ora-db-ddl>` element is not yet defined. The upgrade then adds a `<tablespaces>` element to the `<ora-db-ddl>` element and converts each `<tablespacemapping>` element to an attribute on the `<tablespaces>` element. The upgrade then adds an `<ora-lob>` element to the `<ora-db-ddl>` element and sets the `<ora-lob>` attribute `type` to `BASICFILE`. Although Oracle 12c creates SecureFile LOB columns by default, the configuration upgrade sets the default type for any new LOBs to `BASICFILE` to maintain consistency with the Oracle 11 default. If Oracle LOBs are configured to be SecureFile or compressed SecureFiles, the configuration upgrade does not transfer the DDL settings to `database-config.xml`. These configuration settings must be applied to the new `database-config.xml` database element manually. Note that if you change a DDL configuration, the setting only applies for new objects.

- For SQL Server, the upgrade adds an `<mssql-db-ddl>` element within the `<upgrade>` element of the new configuration, if an `<mssql-db-ddl>` element is not yet defined. The upgrade then adds a `<mssql-filegroups>` element to the `<mssql-db-ddl>` element and converts each `<tablespacemapping>` element to an attribute on the `<mssql-filegroups>` element.

Splitting Localization.xml into Separate Files for each Locale

The upgrade splits the locales from the single `localization.xml` file used in BillingCenter 7.0 into a separate file for each locale defined by a `<GWLocale>` element. The new location for each split `localization.xml` file is `config/locale/locale/`. Each `localization.xml` file can have only one `GWLocale` element in BillingCenter 8.0.

Splitting address-config.xml into Separate Files for each Country

The upgrade splits the address format definitions from the single `address-config.xml` file used in BillingCenter 7.0 into a separate file for each country defined by an `<AddressDef>` element. The new location for each split `address-config.xml` file is `config/geodata/country code/`. Each `address-config.xml` file can have only one `AddressDef` element in BillingCenter 8.0.

Splitting zone-config.xml into Separate Files for each Country

The upgrade splits the zone configuration definitions from the single `zone-config.xml` file used in BillingCenter 7.0 into a separate file for each country. Zones for each country are defined by a `<Zones>` element with a `countryCode` attribute. The new location for each split `zone-config.xml` file is `config/geodata/country code/`. In BillingCenter 8.0 each `zone-config.xml` file can have only one `<Zones>` element that contains zones for a single country.

Splitting currencies.xml into Separate Files for each Currency

The upgrade splits the currency definitions from the single `currencies.xml` file used in BillingCenter 7.0 into a separate file for each currency type. Each currency type is defined by a `<CurrencyType>` element with a `code` attribute. The separate files are each named `currency.xml`. The new location for each `currency.xml` file is `config/currencies/code/`, where `code` is the value of the `code` attribute on the `<CurrencyType>` element.

Moving Country-based Field Validator Definition Files

The upgrade moves each country-based field validator definition file to an individual directory. Country-specific field validator definition files are named with the format `fieldvalidators_country code.xml`, such as `fieldvalidators_JP.xml` for field validators specific to Japan. The upgrade moves each country-specific field validator definition file to `config/fieldvalidators/country code/`. The generic `fieldvalidators.xml` file remains at `config/fieldvalidators/`.

Moving Rules Files up One Directory

The upgrade moves all rules files up one directory from `config/rules/rules/` to `config/rules/`.

Reformatting Rules for Display in Studio Rules Editor

The upgrade reformats `.gr` rule files so that the Studio rules editor recognizes the file contents as rules.

Copying Custom Rules and Adding BillingCenter 8.0.4 Default Rules

The upgrade copies customized rules to the target configuration `modules/configuration/config/rules` directory.

This step also copies the default rules provided with BillingCenter 8.0.4 to a BillingCenter 8.0.4 folder within the `modules/configuration/config/rules` directory of the target configuration. This is so you have a copy of the default rules in a folder in Studio that you can use to compare with your custom rules.

Renaming SOAP Web Services from XML to RWS

The upgrade changes the extension of SOAP web service files in `config/webservices` from `.xml` to `.rws`.

Renaming Plugins from XML to GWP

The upgrade changes the extension of plugin files in `config/plugin/registry` from `.xml` to `.gwp`.

Renaming Display Names Files from XML to EN

The upgrade changes the extension of display names files in `config/displaynames` from `.xml` to `.en`.

Upgrading Display Keys

The upgrade compares display keys from the custom configuration with display keys in the base 7.0 configuration and display keys in the default BillingCenter 8.0 configuration. The following display key files are inspected.

- `display.properties`
- `gosu.display.properties`
- `productmodel.display.properties`
- `studio.display.properties`
- `typelist.properties`

The upgrade compares the case of display property keys in the custom configuration with the case of the key in the default BillingCenter 8.0.4 configuration. If the case does not match, but the value assigned to the key matches the value in the default configuration, the upgrade corrects the case in the custom configuration. If the case of the keys does not match, and the value is different in the custom configuration, the upgrade reports an error.

The upgrade then merges the display keys files into a single file for each locale. This file has the extension `.merged`. The merged display properties files are available in the Configuration Upgrade Tool for comparison with the default BillingCenter `display.properties`. You can merge Guidewire changes and new properties with your custom properties values.

Adding `nullok="true"` to Entity and Extension Foreign Key Columns

The upgrade modifies ETI and EIX files in `config/metadata` and ETX and ETI files in `config/extensions`. The upgrade adds the attribute `nullok="true"` to `<foreignkey>` and `<edgeForeignKey>` elements if the element did not explicitly specify a value for the `nullok` attribute. In BillingCenter 8.0, the `nullok` attribute is required to be explicitly set.

Removing `deletefk` Attribute from Entity and Extension Foreign Keys

The upgrade removes the `deletefk` attribute from all `<foreignkey>` and `<edgeforeignkey>` elements that include a `deletefk` attribute.

Setting XML Namespace on Metadata Files

This step sets the XML namespace on data model and typelist entity and extension files in config/metadata and config/extensions to `http://guidewire.com/datamodel` and `http://guidewire.com/typelists` respectively. You can configure an XML editor to map these namespaces to XSD files that define the structure of data model and typelist files. Map `http://guidewire.com/datamodel` to `BillingCenter/modules/p1/xsd/metadata/datamodel.xsd` and `http://guidewire.com/typelists` to `BillingCenter/modules/p1/xsd/metadata/typelists.xsd`. Then, the XML editor can validate entities as you create or modify them.

The namespace was encouraged but optional prior to 8.0. The namespace must be specified in 8.0.

Upgrading Document Assistant Parameters

In BillingCenter 8.0, Guidewire Document Assistant uses a Java applet deployed using JNLP instead of an ActiveX control. The upgrade updates `config.xml` for this change. In this step, the upgrade replaces legacy Document Assistant ActiveX configuration parameters with the updated ones. The upgrade makes the following changes:

- Renames `AllowActiveX` to `AllowDocumentAssistant`, ignoring the old value. In 8.0 `AllowDocumentAssistant` defaults to `false`, whereas `AllowActiveX` was `true` in prior releases. The deployment, security, and configuration of applets is entirely different from ActiveX controls. Consider Java security issues as part of your decision to deploy the Document Assistant applet.
- Renames `UseGuidewireActiveXControlToDisplayDocuments` to `useDocumentAssistantToDisplayDocuments`, keeping the old value.
- Removes `AllowActiveXAutoInstall`.
- Removes `UseDocumentNameAsFileName`.
- Adds `DocumentAssistantJNLP`.

See “Document Creation and Document Management Parameters” on page 41 in the *Configuration Guide*.

Separating Entities and Typelists

The upgrade creates `entity` and `typelist` folders in config/metadata and config/extensions directories. The upgrade then moves ETI, EIX, and ETX files into the `entity` folders and moves TTI, TIX, and TTX files into the `typelist` folders.

Using the BillingCenter 8.0.4 Upgrade Tool Interface

IMPORTANT Review the automated step descriptions before you proceed. Some automated steps might require you to perform a manual step while merging the configuration. Typically, such automated steps insert a warning into a file. Check the `steps_results.txt` file for warning and error messages. Correct any issues reported. Then, delete `steps_results.txt` and restart the Configuration Upgrade Tool.

After the Configuration Upgrade Tool completes the automated steps, the working area contains up to three versions of the same file:

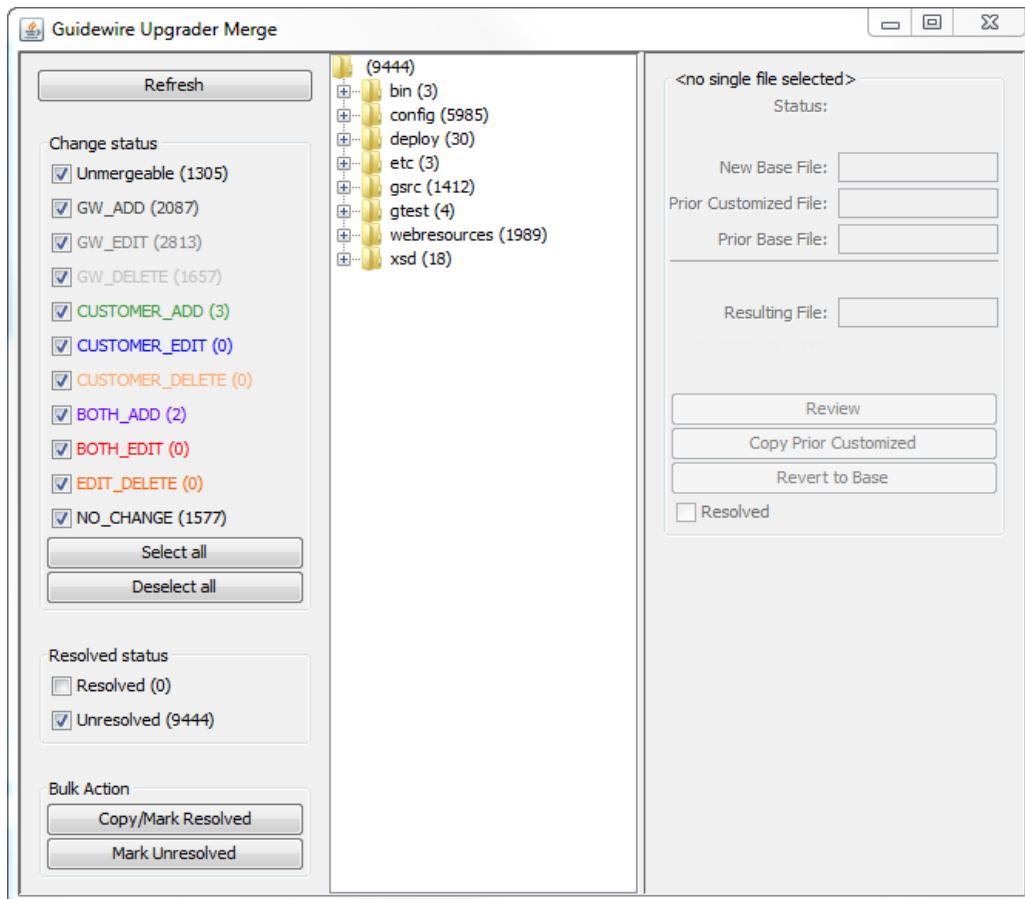
- The `customer` file.
- The `base` file, from which you configured the customer file.
- The `target` file, from BillingCenter 8.0.4.

In the manual process of the upgrade, you decide whether to use one of these versions unchanged, or merge versions together. The Configuration Upgrade Tool provides a user interface to assist with the manual process. This interface has several important functions:

- It shows a complete list of all configuration files.
- It allows you to filter this list. You can, for example, view a list of all files that differ between the target version and your version. See “Change Status Filters” on page 223.
- It displays two or three versions of a file and their differences, using a merge tool you supply, such as Araxis Merge or P4Merge, defined in `upgrade.properties`.
- It lets you edit your file, incorporating changes from the other file versions, and save it.
- It lets you accept this merged version instead of one of the previous versions.
- It lets you edit the file after you have accepted changes from the merge using the text editor defined in `upgrade.properties`.

After you have accepted or merged all files that the Configuration Upgrade Tool displays, the merging process is complete.

The Configuration Upgrade Tool displays three panels. The center panel is a tree view of the files in the configuration, filtered by filter choices selected in the left panel. Files appear in the color of the filter that found them. As you select a file in the center panel, the right panel displays file information and buttons to perform actions on that file.



Filters

The left panel of the Configuration Upgrade Tool contains:

- Refresh Button

- Change Status Filters
- Resolved Status Checkboxes
- Bulk Action Buttons

Refresh Button

If multiple users are working in the same directory, each user can mark files as resolved. The Refresh button refreshes the resolved status of files shown in the Configuration Upgrade Tool for changes contributed by all users working in the same directory.

Change Status Filters

This table lists the change status filters that the Configuration Upgrade Tool displays in the left panel. Use the check boxes next to the filters to select one or any combination of change statuses to view. Use the **Select all** or **Deselect all** buttons to select or deselect all filters. The following table describes change status filters. The Guidewire Action column lists the change Guidewire has made to files matching a status filter since the prior version. The Your Action column lists the change to the file in your implementation:

Merge Status	Guidewire Action	Your Action	Type of change made to file	Action taken by Configuration Upgrade Tool
Unmergeable	change format of file	any	file exists in a different format and thus cannot be merged with an old version	If you resolve the file, the Configuration Upgrade Tool takes no action. The file, in the new format, already exists in the target configuration. The Configuration Upgrade Tool automatically marks certain files as unmergeable, including rules files. The Configuration Upgrade Tool upgrades these files before the interface displays. You can also specify a regular expression pattern in upgrade.properties for file paths to mark files matching that pattern as unmergeable. Set the pattern as the value of the exclude.pattern property.
GW_ADD	add	none	file in target not in base	Typically, you use exclude.pattern to specify source control metadata files. Samples are provided in upgrade.properties for CVS and SVN. If you resolve the file, the Configuration Upgrade Tool takes no action. The file added by Guidewire already exists in the target configuration. Double-clicking opens the file in the text editor specified by upgrader.editor.tool in upgrade.properties.. If you make changes, the tool prompts you to copy the file to the target configuration.
GW_EDIT	edit	none	file in target differs from base	If you resolve the file, the Configuration Upgrade Tool takes no action. The file added by Guidewire already exists in the target configuration. Double-clicking opens the file in the merge tool specified by upgrader.diff.tool in upgrade.properties to perform a comparison between the new Guidewire version and the original base version. If you make changes, the tool prompts you to copy the file to the target configuration.
GW_DELETE	delete	none	file in base not in target	If you resolve the file, the Configuration Upgrade Tool takes no action. The file deleted by Guidewire no longer exists in the target configuration. Double-clicking opens the file in the text editor specified by upgrader.editor.tool in upgrade.properties.. If you make changes, the tool prompts you to copy the file to the target configuration.

Merge Status	Guidewire Action	Your Action	Type of change made to file	Action taken by Configuration Upgrade Tool
CUSTOMER_ADD	none	add	file in customer configuration only	If you resolve the file, the Configuration Upgrade Tool copies the file to the target configuration if the file has not been copied there already. Double-clicking opens the file in the text editor specified by upgrader.editor.tool in upgrade.properties.. If you make changes, the tool prompts you to copy the file to the target configuration.
CUSTOMER_EDIT	none	edit	file differs between customer and base configurations file unchanged between base and target configurations	If you resolve the file, the Configuration Upgrade Tool copies the file to the target configuration if the file has not been copied there already. Double-clicking opens the file in the merge tool specified by upgrader.diff.tool in upgrade.properties to perform a comparison between your custom version and the original base version. If you make changes, the tool prompts you to copy the file to the target configuration.
CUSTOMER_DELETE	none	delete	file exists in the base and target configurations but not in the customer configuration	If you click Delete, the Configuration Upgrade Tool removes the file from the target configuration. If you click, Revert to Base, the Configuration Upgrade Tool leaves the file in the target configuration.
BOTH_ADD	add	add	new file with matching name in both target and customer configurations (rare)	You must either merge the two versions of the file or copy your prior version of the file into the target configuration before you can resolve the file. Double-clicking opens the file in the merge tool specified by upgrader.diff.tool in upgrade.properties to perform a merge between your version and the Guidewire version. If you make changes, the tool prompts you to copy the merged file to the target configuration.
BOTH_EDIT	edit	edit	file changed in both customer and target configurations	You must either merge the two versions of the file or copy your prior version of the file into the target configuration before you can resolve the file. Double-clicking opens the file in the merge tool specified by upgrader.merge.tool in upgrade.properties to perform a three-way merge between your custom version and the updated Guidewire version. If you make changes, the tool prompts you to copy the merged file to the target configuration.

Merge Status	Guidewire Action	Your Action	Type of change made to file	Action taken by Configuration Upgrade Tool
EDIT_DELETE	delete	edit	file changed from base in customer configuration and does not exist in target configuration	<p>If you resolve the file, the Configuration Upgrade Tool takes no action.</p> <p>Double-clicking the file opens your customized file and the original base file in the merge tool specified by <code>upgrader.diff.tool</code> in <code>upgrade.properties</code>. When you close the merge tool, the Configuration Upgrade Tool prompts you to copy the file to the target configuration. If you are sure you want your customized version, you can click Copy prior customized to move the file to the target configuration.</p> <p>The EDIT_DELETE flag appears on a file when your configuration has a customized version of the file but Guidewire has deleted the file from that location. There are two possible reasons for this deletion. One reason is that Guidewire removed the file from BillingCenter. The second reason is that Guidewire has moved the file to a different folder.</p> <p>If Guidewire has completely removed the file, review the <i>BillingCenter New and Changed Guide</i>, release notes, and the Upgrade Diff report for descriptions of the change affecting the deleted file. Then determine if you want to continue moving your customization to the new or changed feature. If not, then the customization will be lost.</p> <p>For the second scenario, find where the file has been moved by searching the target version. Move your customized file to the same location in the working directory and make sure to match any case changes in the filename. When you refresh the list of merge files, the file now appears under the CUSTOMER_EDIT filter. You can now proceed with the merge. If you do not move the file over, you can instead perform the merge manually by opening both files and incorporating the changes.</p>
NO_CHANGE	none	none	file not changed from base configuration in either customer or target configurations	<p>If you resolve the file, the Configuration Upgrade Tool takes no action. The file already exists in the target configuration.</p> <p>Double-clicking opens the file in the text editor specified by <code>upgrader.editor.tool</code> in <code>upgrade.properties</code>. If you make changes, the tool prompts you to copy the file to the target configuration.</p>

Resolved Status Checkboxes

Beneath the change status filters are checkboxes to toggle the visibility of resolved and unresolved files. Use these checkboxes with the change status filters to specify which types of files you want visible in the center panel. For example, you could select **BOTH_EDIT** and **Unresolved** to see files edited in your configuration that have also been updated by Guidewire and are not yet resolved.

The purpose of the resolved status is to have a general idea of the progress you are making in the upgrade. The tool shows the resolved status of the current file (right panel) and the total number of resolved and unresolved files (left panel).

A resolved file is simply a file that you have marked resolved. It does not relate to whether file merging or accepting has occurred.

Bulk Action Buttons

The following buttons in The **Bulk Action** part of the left panel enable you to change the resolved status of a group of selected files:

- **Copy/Mark Resolved**
- **Mark Unresolved**

You can select either one or several files and directories before using these buttons. Use the CTRL key to select multiple files and directories. Selecting a directory selects all files within that directory. You can select all files that match the filters you set by selecting the top-level directory.

After you click **Copy/Mark Resolved**, the Configuration Upgrade Tool opens a dialog detailing the actions it is about to perform.

The tool copies files matching the **CUSTOMER_ADD** and **CUSTOMER_EDIT** filters to the target configuration. If there is already a version of a file in the target configuration, then the tool does not copy the file. A file would be there already if you edited the file and clicked Yes when the tool prompted you to copy the file to the target configuration.

The tool does not do any copying for files matching the **GW_ADD**, **GW_DELETE**, **GW_EDIT**, **NO_CHANGE**, or **Unmergeable** filters. Files matching **GW_ADD**, **GW_EDIT**, **NO_CHANGE**, or **Unmergeable** filters are already present in the target version. Files matching the **GW_DELETE** filter are not in BillingCenter 8.0.4.

You can not bulk resolve multiple files that match the **BOTH_ADD**, **BOTH_EDIT**, or **EDIT_DELETE** filters. Files matching these filters require individual attention. Use the right panel of the Configuration Upgrade Tool to control merging, copying and resolving of these files.

Configuration File Tree

The center panel displays the configuration file tree. Files are color-coded to match filter colors. Files are shown one time, regardless of the number of configurations in which they exist. For information on which configurations a file exists in, select the file and view the right panel. The number of files in each directory that match the selected change status and resolved status filters is shown in parentheses.

File Details Panel

The right panel displays file details for the file you are currently examining, including:

- **Status** – The change status of the file. See “Change Status Filters” on page 223.
- **New base file** – The new version of this file supplied by Guidewire with BillingCenter 8.0.4. If there is not a Guidewire version of this file, such as for **CUSTOMER_ADD**, **EDIT_DELETE** or **GW_DELETE** files, this field is blank.
- **Prior customized file** – The locally customized version of this file from the prior version. If there is not a customized version of this file, such as for **GW_ADD**, **GW_DELETE**, **GW_EDIT** or **NO_CHANGE** files, this field is blank.
- **Prior base file** – The base version of this file in the working directory. If there is not a Guidewire version of this file in the prior base version you are upgrading from, such as for **CUSTOMER_ADD** files, this field is blank.
- **New customized file** – The customized version of this file in the BillingCenter 8.0.4 configuration directory.

The right panel fields are blank if you have multiple files selected.

File Details Panel Actions

The following buttons appear below the file details display in the right panel after you have selected a file:

- **View** – Opens the file in the editor specified in `upgrade.properties`. This button appears for files that are not customized and do not require merging, matching **GW_ADD**, **GW_EDIT**, or **GW_DELETE** filters. Only one of the **View**, **Edit** or **Merge** buttons displays, depending on the file change status.

- **Edit** – Opens the file in the editor specified in `upgrade.properties`. This button appears for custom files that do not require merging, matching the `CUSTOMER_ADD` or `EDIT_DELETE` filters.
- **Merge** – Opens the different versions of the file in the merge tool specified in `upgrade.properties`. This button appears for files that require merging, matching the `BOTH_ADD` or `BOTH_EDIT` filters.
- **Copy prior customized** – Copies the prior customized version of the file to the target configuration. This button is enabled if there is a prior customized version of the file. So it is enabled for files matching `CUSTOMER_ADD`, `CUSTOMER_EDIT`, `BOTH_ADD`, `BOTH_EDIT`, or `EDIT_DELETE` filters.
- **Delete new customized** – Remove the customized version from the target configuration. This reverses the **Copy prior customized** button action. This button is disabled until you have copied a prior customized version of the file into the target configuration.
- **Resolved** – Check this box to label the file resolved. Use the **Resolved** checkbox in the right pane to change the status of a single file. Selecting the **Resolved** checkbox does not copy the file. Use the buttons above this checkbox to handle copying or merging of file versions. You must first unresolve a file before either using the **Delete new customized** action or reapplying changes or merges.

Accepting Files that Do Not Require Merging

The following filters show lists of files that normally do not require merging.

- `CUSTOMER_ADD`
- `CUSTOMER_EDIT`
- `GW_ADD`
- `GW_EDIT`
- `NO_CHANGE`
- `Unmergeable`

You can click the **Copy/Mark Resolved** button in the left panel to resolve groups of these files.

Merging and Accepting Files

Files matching the `BOTH_ADD` and `BOTH_EDIT` filters must be merged before being accepted. Your version must be reconciled with the Guidewire target or base version. In some cases, even if only a single version of the file exists, you might want to look at it before accepting it.

You can use the `pcf.xsd` file in the `modules` directory of the target version to validate merged PCF files.

After you are satisfied with any changes, save the file. This saves the file in a temporary directory. When you close the editor or merge tool, the Configuration Upgrade Tool asks if you want to copy the file to the target configuration. If you click **Yes** (or press `ALT+Y`), the tool copies the file. If you click **No** (`ALT+N`), the tool cancels the popup without copying. The tool always moves files into the target configuration, except if a file is identical to the base or target version. In this case, the tool does not move the file.

Note: Do not edit a file version with `DO_NOT_EDIT` in its file name.

Configuration Merging Guidelines

The first milestone of an upgrade project is to generate the Java and SOAP APIs (by running `gwbc regen-java-api` and `gwbc regen-soap-api`) on the target release. To do this, you must:

- Complete the merge of the data model. This includes all files in the /extensions and /fieldvalidators folders.
- Resolve issues encountered while trying to generate the APIs or start the QuickStart application server.

You can generate the Java and SOAP APIs even if you have errors in your enhancements, rules and PCF files.

Typical errors

- Malformed XML – The merge tool is not XML-aware. There might be occasions in which the file produced contains malformed XML. To check for well-formed XML, use free third-party tools such as Liquid XML, XML Marker, or Eclipse.
- Duplicate typecodes – As part of the merge process, you might have inadvertently merged in duplicate, matching typecodes.
- Missing symmetric relationship on line-of-business-related typelists – You might be missing a parent-child relationship with respect to the line-of-business-related typelist, as a result of merging.

Once the Java and SOAP APIs have been generated, you can begin the work of upgrading integrations.

Second, after you can successfully generate the Java and SOAP APIs, work on starting the server.

You can generate the APIs even if you have errors in your enhancements, rules and PCF files, although error messages will print upon server startup.

Once the server can start on the target release, you can begin the database upgrade process.

Continue with the remainder of the configuration upgrade work, which includes evaluating existing PCF files and merging in desired changes.

Data Model Merging Guidelines

From a purely technical standpoint, not addressing the need to incorporate new features, the following are a few guidelines for merging the data model.

Updating Data Types for Case Sensitivity

Data type definitions are case-sensitive in BillingCenter 8.0. If you are upgrading from an early 7.0 version or a version prior to 7.0, you could have column definitions that specify a type using the wrong case. In this event, the server reports an invalid data type error during startup. If the server reports invalid data type errors, check the case of the type attribute for the column in the ETI or ETX extension file for the entity. Extension files are located in the extensions directory of the configuration module. An ETI file exists for custom entity definitions. An ETX file defines extensions to an entity provided with BillingCenter.

Merging Typelists – Overview

There is no automated process to merge typelists. This is a part of the merge process using the Configuration Upgrade Tool. In general, merge typelists before PCF files.

See the *Upgrade Diffs Report* for an inventory of differences in typekeys between the base release and the target release. To retrieve the *Upgrade Diffs Report* follow the procedure described in “Viewing Differences Between Base and Target Releases” on page 97.

Merge in Guidewire-provided typecodes related to lines of business and retire unused typecodes that you merge in. If you do not include these typecodes, you will have errors in any enhancements, rules, or PCF files that reference the typecode. This also simplifies the process for future upgrades as there will be fewer added line of business typecodes to review.

Pay particular attention if any Guidewire-provided typecodes have the same typecode as a custom version. In this case, modify one of the typecodes so they are unique. Contact Guidewire Support for details.

The Configuration Upgrade Tool displays most typelists you have edited in the CUSTOMER_EDIT filter. If your edits are simply additional typecodes, accept your version.

Use Guidewire Studio to verify PCF files, enhancements, and rules to identify any issues with the files and rules that reference typelists.

Merging Typelists – Simple Typelists

Merge in new typecodes from the target version, BillingCenter 8.0.4. If you do not merge the new typecode, you will have errors in any enhancements, rules, or PCF files that reference the typecode. If you do not want to use a new typecode, retire the typecode by setting the `retired` attribute to `true`.

Merging Typelists – Complex Typelists

A typecode can reference typecode values from another typelist using the `<category>` subelement. If a new typecode references an existing typecode, do not merge the new typecode unless the referenced typecode is retired. Otherwise, you are defining a new relationship. If the referenced typecode is also new, merge in both typecodes. If you do not want to use a new typecode, set the `retired` attribute for the typecode to `true`. The following table summarizes how to handle merging new typecodes that reference other typecodes:

Referenced typecode status	Action
new – exists only in target version	Merge in the new typecode and merge in the referenced typecode in its typelist. If you do not want to use the new typecode, retire it by setting the <code>retired</code> attribute of the typecode to <code>true</code> .
active – exists in base or custom version and is not retired	Do not merge the new typecode.
retired – exists in base or custom version and is retired	Merge in the new typecode. If you do not want to use the new typecode, retire it by setting the <code>retired</code> attribute of the typecode to <code>true</code> .

Reviewing Shared Typekey Configuration

As of version 8.0.3, BillingCenter enforces restrictions on the use of shared typekeys among subtypes.

Same Field Name and Typelist with Different Column

In BillingCenter 7.0 and earlier, if a shared typekey had the same field name and typelist, and specified a different column name, BillingCenter created only one of the typekey columns. The shared typekeys were stored in the single column. As of BillingCenter 8.0.3, if a shared typekey with the same field name and typelist specifies a different column name, BillingCenter creates different columns according to the specification. The database upgrade detects shared typekeys using a single column, creates the additional column, and moves the typekey data to the correct column.

Same Field and Column Names with Different Typelists

In BillingCenter 8.0.1 and 8.0.2, a typekey on subtypes could have the same field name and column name and reference different typelists. As of BillingCenter 8.0.3, this configuration is not allowed. The database upgrade reports an error if it detects this condition.

If you have subtypes with typekeys with the same field and column name that reference different typelists, update your data model configuration to use different column names for each typelist. The database upgrade then moves data to the new column to match the updated data model.

Adding State Typelist Extensions to Jurisdiction

BillingCenter versions 7.0 and newer use a Jurisdiction typelist instead of a State typelist. If your environment includes custom extensions to the State typelist, move those extensions to the Jurisdiction typelist.

To move State typelist extensions to the Jurisdiction typelist

1. Open the `modules/configuration/config/extensions/State.ttx` file in your pre-upgrade starting version in a merge tool.
2. Open `modules/configuration/config/extensions/typelist/Jurisdiction.ttx` file in another panel of the merge tool.
3. Merge typecode elements from `State.ttx` to `Jurisdiction.ttx`.

Merging Entity Extensions

BillingCenter 8.0.4 stores extensions in ETI and ETX files. An `Entity.eti` file defines a new entity. An `Entity.etx` file defines extensions to an existing entity.

Correcting File Naming Issues

In BillingCenter 8.0.4, typelist and entity extension files must be named for the typelist or entity. In versions before 8.0, you could have an extension file name such as `Entity_ABC.etx` or `Typelist_ABC.ttx`. As of BillingCenter 8.0, the file root name must be the entity or typelist name or the entity or typelist name followed by a dot. You can use characters after the root name to include custom name components. For example, `Entity.ABC.etx` is a valid entity extension file name. `Typelist.ABC.ttx` is a valid typelist extension file name. If you have extension files that have names that include characters other than the entity name, rename the files to put the extra characters after a dot.

Correcting Data Type References

BillingCenter entity files must use case-sensitive references to data types. For example, setting a `<column-override>` to have `type="shorttext"` is not the same as setting `type="ShortText"`. In this case, the former is valid while the latter is not.

Review each entity extension you have added to make sure data type references are set with the correct case.

To review and correct extension data type references

1. In Studio, expand `configuration → config → Extensions → Entity`.
2. Double-click each ETX file. If the file has an invalid data type reference, Studio reports that the extension field overrides validator detected a column override that refers to a non-existent data type.
3. For any such errors, select the column. Then select the correct case-sensitive `Value` for the `type` from the drop-down list.

Reviewing Optional Indexes

Guidewire often adds indexes to entities in the target configuration to improve the performance of database queries in BillingCenter 8.0.4. BillingCenter requires some of these indexes. Guidewire adds required indexes to entity definitions in the data model. Other indexes are recommended for most installations but can be disabled if they negatively impact performance. Guidewire adds optional indexes to entity extensions so you can disable any of these indexes if necessary.

Use the Configuration Upgrade Tool to resolve extension files. When you merge your custom extensions with Guidewire changes, review each new index added by Guidewire. In most cases, include the new index in the merged extension file. You can modify or remove index definitions based on usage in your deployment.

Updating `setterScriptability` Attributes

The `setterScriptability` attribute can no longer be set to `external` as of BillingCenter 8.0. For any instances you have of the attribute `setterScriptability` set to `external`, change the value to `all`.

Reviewing Custom Extensions

Generate and review the data dictionary for the target version to identify any custom extensions that are now obsolete due to Guidewire adding a similar field to the base BillingCenter.

To generate the data dictionary

1. From the command line, navigate to the `bin` directory of the target version.
2. Run the command `gwbc regen-dictionary`.

This command generates the data and security dictionaries in the `build/dictionary` directory of the target version. To view the data dictionary, open `build/dictionary/data/index.html` in a web browser.

Compare the target version data dictionary with the version in your current environment. If you have extensions that are now available as base fields, consider migrating the data in those fields to the base version. Consider whether an extension is still on the appropriate entity. A new entity could be a more appropriate location for the extension. Review key data model changes that might impact your custom extensions.

If you change an extension location or migrate to a new base field, update any PCF, rule or library that references the extension to reference the new location.

Reconciling the Database with Custom Extensions

Extensions defined in ETI and ETX files must match the physical database. Delete all physical columns in the database that are not part of the base installation or defined as extensions before starting the server.

Removing Obsolete Attributes

Guidewire has removed the `deletefk` and `onDelete` attributes of the `<foreignkey>` and `<edgeForeignKey>` elements. These attributes were deprecated in an earlier major version. Now that the attributes are removed from the schema for entity definition files, if the attributes are listed, the server reports an error and does not start. Remove any occurrences of `deletefk` and `onDelete` attributes from `<foreignkey>` and `<edgeForeignKey>` elements in custom entities.

Updating Extractable Edge Foreign Keys

Guidewire has removed the `<implementsEntity>` element from `<edgeForeignKey>` and `<edgeForeignKey-override>`. In BillingCenter 8.0, to make an edge foreign key extractable, set the Boolean `extractable` attribute on the element to `true`.

For any extractable edge foreign keys and edge foreign key overrides, delete the `<implementsEntity>` element from the key definition. Then add the attribute `extractable="true"` to the `<edgeForeignKey>` or `<edgeForeignKey-override>` element.

Converting Money to MonetaryAmount

BillingCenter upgrade cannot automatically convert the `Money` data type to the `MonetaryAmount` data type. If you created entity extensions, the upgrade process will not upgrade your extensions that include properties that use the `Money` data type.

Before you upgrade, manually update any extension properties that use the `Money` data type.

Define the `MonetaryAmount` property as follows:

- The name of the new `MonetaryAmount` property is the same as the name of the `Money` property
- If the old `Money` property had a `columnName` attribute defined as something other than the `Money` property name, use that old `Money.columnName` as the name of the new `MonetaryAmount.amountColumnName` attribute.
- For extensions to entities that will be `InCurrencySilo` entities in BillingCenter 8, you must add a `DefaultValueZero` tag if your old `Money` property had the `default` attribute set to 0.
- Set `scaleToCurrency` to `true` unless you have a requirement to do otherwise.
- Set the `soapNullOk` attribute to `true`

If you used an extension column to represent money, but did not set the column to the `money` datatype, contact Guidewire Support.

The following examples show how you must redefine `Money` properties in your extensions to `MonetaryAmount` properties before you proceed with upgrade:

Old Total:

```
<column
  name="Total"
  type="money"/>
```

New Total:

```
<monetaryamount
  name="Total"
  amountColumnName="Total"
  soapNullOk="true" />
```

Old Total with default = 0:

```
<column
  name="Total"
  default="0"
  type="money"/>
```

New Total:

```
<monetaryamount
  name="Total"
  amountColumnName="Total"
  soapNullOk="true" />
  <tag name="DefaultValueZero"/>
</monetaryamount>
```

Old Total where name and columnName differ:

```
<column
  name=" Total"
  columnName="totalColumn"
  type="money"/>
```

New Total:

```
<monetaryamount
  name="Total"
  amountColumnName="totalColumn"
  soapNullOk="true" />
```

Preserving Payment Method Details

The database upgrade deletes payment method details columns from the following entities:

- Account
- Disbursement
- IncomingProducerPayment
- OutgoingPayment
- PaymentMoneyReceived
- PaymentRequest
- Producer
- SuspensePayment

From these entities, the database upgrade deletes the following payment method details columns:

- BankABANumber
- BankAccountNumber
- BankAccountType
- BankName
- CreditCardExpDate
- CreditCardIssuer
- CreditCardNumber
- PaymentMethod

See “Converting to Payment Instruments” on page 286 for details.

During upgrade of the database, the `PaymentMethod` is converted into a `PaymentInstrument` and stored appropriately. However, you might want to preserve the other payment method details columns.

There are two options that you can use to preserve payment method details information. You can define the payment method details columns as extensions in the BillingCenter 3.0 and 8.0.4 data model. Or, you can keep a copy of your BillingCenter 3.0 database, and match BillingCenter 8.0 payment instruments to the BillingCenter 3.0 payment method details. The following procedures explain these options.

To preserve payment method details information in the BillingCenter 8.0 database

1. In Studio for the pre-upgrade BillingCenter 3.0.x environment, expand `Data Model Extensions` → `metadata` → `bc`.
2. If you do not have an extension file for the entity, right-click the base entity name, such as `Account.eti`, and select `Create extension file`. Studio creates a basically empty extension file named `entity.etx`, places it in the `Data Model Extensions` → `extensions` folder, and opens it in a view tab for editing. If you already have an extension file for the entity, select the extension file from the `Data Model Extensions` → `extensions` folder.
3. Add the payment method details columns other than `PaymentMethod` to the extension. Append the suffix `_EXT` to each extension name to distinguish the extension from BillingCenter 8.0.4 base columns. Note that the `CreditCardIssuer` typelist was removed in BillingCenter 7.0, so the `CreditCardIssuer` column is defined in the extension as a `shorttext` column rather than a typekey.

```
<?xml version="1.0"?>
<extension xmlns="http://guidewire.com/datamodel" entityName="Account">
  <column
    desc="Bank ABA Number"
    name="BankABANumber_EXT"
    type="shorttext">
    <columnParam
      name="encryption"
      value="true"/>
  </column>
  <column
    desc="Bank Account Number"
    name="BankAccountNumber_EXT"
    type="shorttext">
    <columnParam
      name="encryption"
      value="true"/>
```

```

</column>
<column
  desc="Bank Name"
  name="BankName_EXT"
  type="shorttext"/>
<column
  desc="Credit Card Expiration Date"
  name="CreditCardExpDate_EXT"
  type="dateonly"/>
<column
  desc="Credit Card Number"
  name="CreditCardNumber_EXT"
  type="shorttext">
  <columnParam
    name="encryption"
    value="true"/>
</column>
<typekey
  desc="The account's bank account type, if applicable"
  name="BankAccountType_EXT"
  typelist="BankAccountType"/>
<column
  desc="Credit Card Issuer"
  name="CreditCardIssuer_EXT"
  type="shorttext"/>
</extension>

```

4. Click File → Save Changes.
5. Repeat step 2 through step 4 for the remaining entities that have payment method details columns deleted by the database upgrade.
6. Start the BillingCenter 3.0 server. BillingCenter adds the new extension columns to the underlying tables.
7. Copy the original payment method details to extended columns. For example, to copy data in bc_account, execute the following SQL:


```

UPDATE a
SET a.BankABANumber_EXT = a.BankABANumber,
    a.BankAccountNumber_EXT = a.BankAccountNumber,
    a.BankAccountType_EXT = a.BankAccountType,
    a.BankName_EXT = a.BankName,
    a.CreditCardNumber_EXT = a.CreditCardNumber,
    a.CreditCardIssuer_EXT = a.CreditCardIssuer,
    a.CreditCardExpDate_EXT = a.CreditCardExpDate
FROM bc_account a
      
```
8. Repeat step 7 with the remaining tables. Note that the PaymentMoneyReceived entity is a subtype of BaseMoneyReceived and is defined on the bc_BaseMoneyReceived table, so adjust your SQL command accordingly.
9. When merging the data model configuration files, include the payment method details extensions that you just created so that the extensions will be included in the BillingCenter 8.0.4 database.

To match BillingCenter 8.0 payment instruments to 3.0 payment method details

1. Save a copy of the BillingCenter 3.0 database. You will not upgrade this copy. Payment method details are found in the following tables:
 - bc_Account
 - bc_Producer
 - bc_PaymentRequest
 - bc_SuspensePayment
 - bc_Disbursement
 - bc_OutgoingPayment
 - bc_BaseMoneyReceived
 - bc_IncomingProducerPayment
2. Upgrade a different copy of your BillingCenter 3.0 database to BillingCenter 8.0.4.

3. Match payment instruments to payment method details. With the saved BillingCenter 3.0 database (in examples below, named `beforeUpgradeDB`) and upgraded database (`postUpgradeDB`), the following SQL associates a payment instrument to payment details. This data can be saved elsewhere in an external system and therefore provide a token back to the existing payment instrument.

For bc_Account:

```
SELECT a.PublicID, a.PaymentMethod,
       b.ID, b.AccountNumber, b.BankABANumber, b.BankAccountNumber, b.BankAccountType,
       b.BankName, b.CreditCardNumber, b.CreditCardIssuer, b.CreditCardExpDate
  FROM postUpgradeDB.dbo.bc_paymentinstrument a, beforeUpgradeDB.dbo.bc_account b
 WHERE a.AccountID IS NOT NULL AND a.AccountID = b.ID
```

For bc_Producer:

```
SELECT a.PublicID, a.PaymentMethod,
       b.ID, b.Name, b.BankABANumber, b.BankAccountNumber, b.BankAccountType, b.BankName,
       b.CreditCardNumber, b.CreditCardIssuer, b.CreditCardExpDate
  FROM postUpgradeDB.dbo.bc_paymentinstrument a, beforeUpgradeDB.dbo.bc_producer b
 WHERE a.ProducerID IS NOT NULL AND a.ProducerID = b.ID
```

For bc_BaseMoneyReceived, bc_Disbursement, bc_IncomingProducerPayment, bc_OutgoingPayment, bc_PaymentRequest, or bc_SuspensePayment:

```
SELECT a.PublicID, a.PaymentMethod,
       b.ID, b.Amount, b.BankABANumber, b.BankAccountNumber, b.BankAccountType, b.BankName,
       b.CreditCardNumber, b.CreditCardIssuer, b.CreditCardExpDate
  FROM postUpgradeDB.dbo.bc_paymentinstrument a, beforeUpgradeDB.dbo.savedTable b,
       postUpgradeDB.dbo.savedTable c
 WHERE c.PaymentInstrumentID = a.ID AND c.ID = b.ID
```

Replace `savedTable` with the table name in the saved BillingCenter 3.0 database, such as `bc_Disbursement`.

Migrating BCContact and PaymentDetails Extensions

Before you upgrade, remove any extension field definitions from `BCContact.etx` and `PaymentDetails.etx` and add those extension fields to the corresponding supported entity. Move any extension field definitions from `BCContact.etx` to `Contact.etx`. Move any extension field definitions from `PaymentDetails.etx` to `PaymentReceipt.etx`. The extension field names and definitions must not change.

For details on the changes that the database upgrade makes to contacts, see “Converting and Removing BCContact Subtype” on page 280.

Changes to the Logging API

Guidewire updated the logging API between BillingCenter 7.0.1 and 7.0.2. Although changes to logging infrastructure were extensive, the purpose of these changes is simplification of logging usage. This document describes changes to the Guidewire logging API. If you are upgrading from a version prior to BillingCenter 7.0.2, use this section as a guide to update your configuration files to the new logging API.

Conceptual Changes to Logging

Old API

<code>com.guidewire.logging.Logger</code>	The <code>Logger</code> class implements all logging functions. Instantiate the class with a new statement. This class is a wrapper around the Log4J <code>Logger</code> class.
<code>com.guidewire.logging.LoggerFactory</code>	The <code>LoggerFactory</code> class has two purposes. First, the class instantiates the logging infrastructure and determines the logging configuration. Second, the class is a factory that produces <code>Logger</code> instances.
<code>com.guidewire.logging.LoggerCategory</code>	The <code>LoggerCategory</code> class is a subclass of the <code>Logger</code> class. An instance of the <code>LoggerCategory</code> class behaves exactly the same way as instance of <code>Logger</code> , but <code>LoggerCategory</code> also maintains a set of static members, which are predefined loggers.
<code>com.guidewire.xx.system.logging.XXLoggerCategory</code> in which xx is a product-specific code such as bc, cc, or pc.	The <code>XXLoggerCategory</code> classes are application-specific subclasses of <code>LoggerCategory</code> . Normally, application-specific <code>LoggerCategory</code> classes maintain additional static <code>Logger</code> members for applications to use.

New API

<code>gw.p1.logging.Logger</code>	<p>Logger was converted from a class to an interface in BillingCenter 7.0.2. In 8.0, <code>Logger</code> is deprecated. You can update code to use <code>org.slf4j.Logger</code> instead of <code>gw.p1.logging.Logger</code>.</p> <p>The <code>Logger</code> interface provides all necessary functionality and hides implementation. This <code>Logger</code> interface explicitly prohibits certain functions that the previous <code>Logger</code> class allowed:</p> <ul style="list-style-type: none"> • You cannot set logging level within the application • You cannot add nor remove appenders within the application <p>The purpose of the <code>Logger</code> interface is to log application-specific messages. Every application component must use its own <code>Logger</code> instance to log messages relevant to the component itself.</p> <p>Do not perform logger management from within the component, such as defining the logging level for a logger. Instead, use the <code>logging.properties</code> file and the application interface to control logging levels and appenders.</p>
<code>gw.p1.logging.LoggerFactory</code>	<p>The <code>LoggerFactory</code> class retains its original functionality, but some methods have changed.</p> <p>This <code>LoggerFactory</code> has two purposes. First, the class instantiates the logging infrastructure and determines the logging configuration. Second, the class is a factory that produces <code>Logger</code> instances.</p>
<code>gw.api.util.Logger.forCategory</code>	<p>The <code>forCategory</code> method of the <code>Logger</code> class returns a <code>Logger</code> for the category, which is passed as a parameter to the <code>forCategory</code> method.</p>
<code>com.guidewire.xx.system.logging.XXLoggerCategory</code> in which xx is a product-specific code such as bc, cc, or pc.	<p>The <code>XXLoggerCategory</code> classes are application-specific subclasses of <code>LoggerCategory</code>. They retain their function of maintaining additional static <code>Logger</code> members for applications to use, but the static members now are instances of the <code>Logger</code> interface. You can no longer instantiate application-specific subclasses of <code>LoggerCategory</code>.</p>
<code>gw.api.system.XXLoggerCategory</code> in which xx is a product-specific code such as bc, cc, or pc.	<p>A mirror class to expose the logger category. It inherits all loggers defined in its <code>gw.p1</code> counterpart.</p>

Instantiating Loggers

Old API

With the old API, you can instantiate logger instances using `Logger`, `LoggerCategory`, `LoggerFactory` or an instance of `LoggerCategory`. Any of the following statements instantiates a logger:

```
Logger logger1 = new Logger("Logger1");
Logger logger12 = new Logger(logger1, "Sublogger2");
LoggerCategory category1 = new LoggerCategory("Category1");
LoggerCategory category12 = new LoggerCategory(category1, "Subcategory2");
Logger factoryLogger1 = LoggerFactory.getInstance().getLogger("FactoryLogger1");
Logger factoryLogger12 = LoggerFactory.getInstance().getLogger(factoryLogger1,"Sublogger2");
Logger apiLogger = LoggerCategory.API;
LoggerCategory apiCategory = LoggerCategory.API;
Logger apiSubLogger = new LoggerCategory(LoggerCategory.API, "WebAPI");
LoggerCategory apiSubCategory = new LoggerCategory(LoggerCategory.API, "WebAPI");
```

New API

With the new API, you work only with instances of the `Logger` interface. You can no longer directly instantiate logger instances, so the new API supports only a few methods to obtain a logger instance:

```
// Using gw.* package
import gw.util.*;
import gw.api.system.*;
import gw.api.util.*;

ILogger apiLogger = PLLoggerCategory.API;
ILogger apiSubLogger = Logger.forCategory.(PLLoggerCategory.API, "WebAPI");

// Using com.guidewire.* package
import com.guidewire.logging.*;
Logger logger1 = LoggerFactory.getLogger("Logger1");
Logger logger12 = LoggerFactory.getLogger(logger1, "Sublogger2");
Logger apiLogger = LoggerCategory.API;
Logger apiSubLogger = LoggerFactory.getLogger(PLLoggerCategory.API, "WebAPI");
```

The new API loses no functionality compared with the old API, but fewer arbitrary options exist.

Note: The `LoggerFactory` class no longer has a `getInstance()` method. The `LoggerFactory.getLogger()` method is now static.

Logging Messages

After you obtain an instance of the `Logger` with the new API, you can use the same methods as the old API to log messages. However, a new interface also allows SLF4J formatting of the messages.

Old API

```
logger.info("Started application " + appName + " with parameters " + parms.toString());
logger.info("Listening to the port " + Integer.toString(portNumber));
```

New API

```
if (wantOldStyle) {    // Old style
    logger.info("Started application " + appName + " with parameters " + parms.toString());
    logger.info("Listening to the port " + Integer.toString(portNumber));
} else {                // New style
    logger.info("Started application {} with parameters {}", appName, parms.toString());
    logger.info("Listening to the port {}", new Integer(portNumber));
}
```

Passing Loggers as Parameters

With the new API, the `LoggerCategory` class exists and has static members. However, those members are instances of the `Logger` interface instead of the `LoggerCategory` itself.

Old API

```
private LoggerCategory getApiLogger() {  
    return LoggerCategory.API;  
}  
  
// ...  
LoggerCategory myLogger = getApiLogger();  
myLogger.debug("...");
```

New API

```
// Using gw.* package.  
private gw.pl.logging.Logger getApiLogger() {  
    return PLLoggerCategory.API;  
}  
// ...  
gw.pl.logging.Logger myLogger = getApiLogger();  
myLogger.debug("...");  
  
// Using SLF4J.  
private org.slf4j.Logger getApiLogger() {  
    return PLLoggerCategory.API;  
}  
  
// ...  
org.slf4j.Logger myLogger = getApiLogger();  
myLogger.debug("I am Logger {}", myLogger.toString());
```

Changes to Iterators in PCF Files

As of BillingCenter 7.0, PCF widgets explicitly identify the iterator for which they apply. Explicit iterator references enable referencing widgets in other PCF files and increase the speed of PCF verification.

The following widgets can expose contained iterators:

- ListViewPanel
- ListDetailPanel
- PanelSet
- PanelRow
- RowSet

In Studio, widgets that can expose a contained iterator have a new **Exposes** tab that lists any exposed iterators.

If a panel exposes iterators and is also modal, it is possible that not all modes have the same iterators defined. In this case the iterators must still be exposed, but the applicable property must be set to `false`.

Guidewire updated many PCF files in order to explicitly identify iterators.

For custom PCF files, Guidewire provides a utility to upgrade PCF files so that widgets explicitly define the iterator to which they apply. See “Running PCF Iterator Upgrade” on page 246.

Updating Namespace on Files Loaded by GX Models

The namespace used by GX model schemas has been updated to take into account the parent package of the model. Any files that are imported using GxModels must be updated to use the correct namespace. For example, the namespace for AddressModel.gx was defined as `xmlns="http://guidewire.com/pc/gx/AddressModel.gx"`. The namespace is now defined as `xmlns="http://guidewire.com/pc/gx/gw.webservice.pc.pc800.gxmodel.addressmodel"`.

Adding DDL Configuration Options to database-config.xml

The configuration upgrade includes an automated step to move the database configuration from config.xml to database-config.xml. The automated step transforms most of the configuration to the 8.0 standard. However, the automated step does not transform certain DDL-related configuration settings. If you have DDL-related configuration settings for compression, Oracle SecureFile LOBs, or partitioning, recreate the configuration in the database-config.xml file.

DDL configuration setting changes only apply to new objects. For example, if you change an existing table from BasicFile to SecureFile LOBs, only new LOB columns will be SecureFile LOBs.

For instructions, see the following topics:

- “Configuring Compression” on page 24 in the *Installation Guide*
- “Configuring BillingCenter to Use Oracle SecureFile LOBs” on page 31 in the *Installation Guide*
- “Configuring Table Partitioning for Oracle” on page 32 in the *Installation Guide*

Merging Changes to Field Validators

The <ValidatorDef> element in fieldvalidators.xml accepts new attributes with BillingCenter 8.0. All of the new attributes are optional. These attributes, the values that you can set the attributes to, and the default value of the attributes are listed in the following table:

Attribute	Values	Default	Description
validation-level	none relaxed strict	strict	The validation-level is passed to the Gosu validators. The functionality for each validation level is specific to the custom validator.
validation-type	gosu regex	regex	If validation-type is set to regex, the value of the <ValidatorDef> defines a regular expression that BillingCenter uses to validate data entered into a field that uses the field validator. If the validation-type is set to gosu, the value of the <ValidatorDef> is a Gosu class. The Gosu class must extend FieldValidatorBase and override the validate method. See gw.api.validation.PhoneValidator for an example. Ensure that any Gosu validators that you define are low-latency for performance reasons.

Guidewire has updated some `<ValidatorDef>` elements in `fieldvalidators.xml` to use the new attributes. For `<ValidatorDef>` elements that you have customized, review the use of the new attribute to see if the behavior is what you want. For `<ValidatorDef>` elements that you have not customized, you can accept the new attributes.

Renaming PCF files According to Their Modes

In BillingCenter 8.0, a PCF file may contain only a single mode, and must include the name of its mode, if any, in the file name. Violations of this rule produce compilation errors in Guidewire Studio. For example, if a file `MyFileDV.pcf` had previously defined two modes, abc and xyz, those modes must now be split into separate files, named `MyFileDV.abc.pcf` and `MyFileDV.xyz.pcf`. Even if a PCF file only contains a single mode, but that mode is not included in the file name, you must still rename the file to include the mode.

Guidewire has renamed all PCF files included in the default 8.0 configuration. However, the Configuration Upgrade Tool might not automatically fix some of your own added or changed files. In particular, take notice of `EDIT_DELETE` conflicts during the three-way merge process. Guidewire could have renamed or split apart the file based on its PCF modes rather than deleted the file. In that case, the new PCF file or files are likely to be in the same directory. Merge your changes into the new file or files.

Updating Rounding Mode Parameter

BillingCenter 8.0.4 has changed from using the `RoundingMode` parameter to the `DefaultRoundingMode` parameter.

Check that you either have no usages of `DefaultRoundingMode` in your configuration, including Gosu code and PCF files, or that if used, it has the same value as `RoundingMode`.

Then, update all usages of `RoundingMode` to use `DefaultRoundingMode`. Define the `DefaultRoundingMode` parameter in `config.xml` to reflect the previous `RoundingMode` parameter value. Change the `RoundingMode` param element in `config.xml` by changing the name to `DefaultRoundingMode` and removing `ROUND_` from the front of the value

For example, change:

```
<param name="RoundingMode" value="ROUND_HALF_EVEN"/>
```

to:

```
<param name="DefaultRoundingMode" value="HALF_EVEN"/>
```

Finally, before the database upgrade, run the following SQL:

```
DELETE FROM bc_systemparameter WHERE name = 'config_param_DefaultRoundingMode'
```

Merging compatibility-xsd.xml

The BillingCenter 7.0 Configuration Upgrade Tool generates a `compatibility-xsd.xml` file which enables existing 3.0 code referencing XSD types to continue compiling. BillingCenter 8.0 includes its own `compatibility-xsd.xml` file. Therefore, the `compatibility-xsd.xml` file appears under the `BOTH_ADD` filter in the Configuration Upgrade Tool.

Merge all XSD entries in the two files together. Entries in the BillingCenter 8.0 `compatibility-xsd.xml` file are required to keep base code compiling. For example, where ClaimCenter code references XSD types related to ISO web services. For custom XSDs, the XSD entry will probably maintain backwards compatibility and allow the code to compile.

However, because Gosu is case sensitive in 8.0, the code might still not compile if the prior version code was not referencing XSD types with the proper case.

If you have to edit usages of the XSD types anyway, Guidewire recommends that you remove the XSD entry in `compatibility-xsd.xml`, and fix any case issues. By doing that, the code will now compile against the most recent version of the XSD typeloader, instead of relying on backwards-compatible behavior which might disappear later.

For example:

A custom environment adds a `Custom.xsd` file to the `gw` package directory. The XSD file contains a root element called `root`.

The 3.0 XSD typeloader exposes this element to Gosu as `gw.Custom.root`. Add `gw.Custom` to `compatibility-xsd.xml` to maintain this behavior.

The 7.0 and later XSD typeloader exposes this element to Gosu as `gw.custom.Root`.

Code that referenced the type as `gw.custom.root` would have compiled in 3.0 and 7.0 because Gosu was case-insensitive. Now that Gosu in 8.0 is case-sensitive for all types except entity types, this code will not compile with either behavior without fixing the capitalization. Guidewire recommends choosing the 7.0 and later behavior and fixing the capitalization of your code to match.

Merging Display Properties

The Configuration Upgrade Tool updates display properties files, such as `display.properties`, as described in “Upgrading Display Keys” on page 220 to create a merged file with the extension `.merged`. You could have conflicts in the files if you have a different number of parameters for a key than the 8.0 version or if you have a different value.

If the number of parameters differs from the 8.0 version, match your parameter set to the 8.0 version of the key.

If the value is different, choose which value you want to use in your BillingCenter configuration.

Merge changes into `display.properties.merged`. When you save the file, the Configuration Upgrade Tool saves it to the configuration module without the `.merged` extension.

If you have added locales, you can export a full list of display keys and typelists from the default BillingCenter 8.0.4 locale to any locale you have defined. This list includes a section for display keys and typelists that do not yet have values defined for your locale. You can use this list to determine which display keys and typelists require localized values. You can then specify those values and import the list. See “Translating New Display Properties and Typecodes” on page 246.

In BillingCenter 8.0, Studio trims trailing spaces from display keys by default. You can modify this behavior using the following procedure:

1. Click File → Settings.
2. Under IDE Settings click Editor.
3. Under Other, change the value of Strip trailing spaces on Save to None.
4. Click OK.

Merging Other Files

In some cases, the differences between files cannot be merged effectively using a comparison tool. In particular, `config.xml`, `logging.properties`, and `scheduler-config.xml` often have many changes between major versions. Consider adding your custom changes to the new Guidewire-provided version instead of merging from prior versions if the presentation of these files in the merge tool is too daunting.

During startup, BillingCenter 8.0.4 reports a warning message if you have configuration parameters defined in `config.xml` that BillingCenter 8.0.4 does not use. BillingCenter ignores any unused parameters. You might have old parameters in `config.xml` that BillingCenter does not use. If BillingCenter 8.0.4 reports that there are unknown parameters specified, remove these parameters from `config.xml`.

If your installation contains a language that is not one of the core Guidewire-supported languages in the base configuration, in `config.xml` copy the value of `DefaultApplicationLocale` to `DefaultApplicationLanguage`. The core Guidewire-supported languages in the base configuration are U.S. English, Italian, German, Spanish, French, Chinese, and Japanese.

Migrating to 64-bit IDs During Upgrade (SQL Server Only)

BillingCenter 8.0.4 uses 64-bit `BIGINT` primary key and foreign key identifiers and `datetime2` date columns. BillingCenter versions prior to 7.0 used 32-bit signed `INTEGER` primary key and foreign key identifiers, which do not provide as many unique identifiers, and `datetime` date columns.

Converting all primary key and foreign key identifiers to 64-bit and all date columns to `datetime2` is a time-intensive process. So, BillingCenter 8.0.4 provides the configuration parameter `MigrateToLargeIDsAndDatetime2` to enable the migration. By default `MigrateToLargeIDsAndDatetime2` is set to `false`, so you can complete the database upgrade and then perform the migration later. This process is described in “Migrating to 64-bit IDs After Upgrade (SQL Server Only)” on page 319.

If you instead want to perform the migration during the database upgrade, follow the procedure in this topic.

Guidewire recommends that you eventually complete the migration to 64-bit primary key and foreign key identifiers to ensure that there are enough unique identifiers available for BillingCenter.

The `MigrateToLargeIDsAndDatetime2` parameter controls the migration of primary key and foreign key identifiers only. The database upgrade automatically converts some other columns defined as the `longint` data type to `BIGINT`. The database upgrade performs that conversion regardless of the setting of `MigrateToLargeIDsAndDatetime2`.

To migrate primary key and foreign key identifiers and date columns during the upgrade

1. Open Studio for BillingCenter 8.0.4.
2. Open `configuration` → `config` → `config.xml`.
3. Change the value of the `MigrateToLargeIDsAndDatetime2` parameter to `true`.
4. Save your changes.
5. Open `configuration` → `config` → `database-config.xml`.
6. Add the `allowUnloggedOperations` attribute to the `upgrade` element.
`<upgrade allowUnloggedOperations="true" ... />`
If you do not require full logging, set `allowUnloggedOperations` to `true` to improve performance of the conversion. Set the SQL Server recovery model to Simple or Bulk logged.
If you do require full logging due to the presence of solutions such as Database Mirroring, continue to use the Full recovery model and set `allowUnloggedOperations` to `false`.

See “Disabling SQL Server Logging” on page 274.

7. Save your changes. BillingCenter will perform the migration to 64-bit BIGINT identifiers and datetime2 date columns during the upgrade when you start the server.

Fixing Gosu Issues

Review additions and changes to Gosu code in the BillingCenter New and Changed Guide. Update your Gosu code for these changes. Use the procedures in this topic to detect and fix these issues.

See also

- “New and Changed in Gosu in 8.0” on page 47 in the *New and Changed Guide*
- “What’s New and Changed in 8.0 Maintenance Releases” on page 17 in the *New and Changed Guide*

Gosu Case Sensitivity

BillingCenter 8.0 has strict case-sensitivity for Gosu code.

To detect and fix case-mismatch issues

1. Right-click a folder in the Project pane and select **Analyze → Run Inspection by Name....**

Note: Do not select the whole project as the inspection is resource intensive.
2. Enter case and double-click **Name is referenced with improper case**.
3. In the dialog, set **Inspection scope** to **Directory**.
4. Deselect **Include test sources**.
5. Click **OK**.
6. In the Results pane, expand **Case mismatch issues**, if present.
7. Right-click the **Name is referenced with improper case** issue type, and click **Apply Fix ‘Case mismatch issues’**.
8. Click the **Save All** icon.
9. Repeat this procedure for the selected folder until no case mismatch issues are reported or the count stops dropping. It might not drop all the way to zero. Keep a record of any folders that do not reach zero errors.
10. Continue this process for all folders containing files with Gosu code.
11. If any folders have an error count above zero, and the count is not dropping after you apply the fix, compile the project to detect other errors.

Inequality Operator

The inequality operator `<>` is no longer valid and must be replaced with `!=`.

To detect and fix the obsolete inequality operator

1. Right-click a folder in the Project pane and select **Analyze → Run Inspection by Name....**

Note: Do not select the whole project as the inspection is resource intensive.
2. Enter `The <>` and double-click `The <> operator is obsolete`.
3. In the dialog, set **Inspection scope** set to **Directory**.
4. Click **OK**.

5. In the Results pane, expand **Equality issues**, if present.
6. Right-click issue type **The <> operator is obsolete**, and click **Apply Fix 'Equality issues'**.
7. Click the **Save All** icon.
8. Repeat this procedure for the selected folder until no equality issues are reported or the count stops dropping. It might not drop all the way to zero. Keep a record of any folders that do not reach zero errors.
9. Continue this process for all folders containing files with Gosu code.
10. If any folders have an error count above zero, and the count is not dropping after you apply the fix, compile the project to detect other errors.

Ambiguous Method Calls

Previous versions of Gosu reported a warning on ambiguous method calls. Ambiguous method calls can hide a logical bug in your code. Previously, the Gosu compiler selected the best matching method to remove ambiguity. For BillingCenter 8.0, ambiguous calls are now an error instead of a warning. Studio now has a code inspection to identify and optionally fix any ambiguous code to previous Studio behavior. This inspection is disabled by default. To find and fix potential logical errors, Guidewire recommends that you run the inspection and carefully individually analyze every ambiguous call before applying any proposed fix.

To detect and fix ambiguous method calls

1. Right-click a folder in the Project pane and select **Analyze → Run Inspection by Name....**
- Note:** Do not select the whole project as the inspection is resource intensive.
2. Enter **The method** and double-click **The method call is ambiguous, it can be fixed by adding casts**.
3. In the dialog, set **Inspection scope** to **Directory**.
4. Deselect **Include test sources**.
5. Click **OK**.
6. In the Results pane, expand **The method call is ambiguous, it can be fixed by adding casts**, if present.
7. Analyze and fix any ambiguous method calls that are reported.
8. Repeat this procedure for the selected folder until no ambiguous method call issues are reported or the count stops dropping. It might not drop all the way to zero. Keep a record of any folders that do not reach zero errors.
9. Continue this process for all folders containing files with Gosu code.
10. If any folders have an error count above zero, and the count is not dropping after you apply the fix, compile the project to detect other errors.

Nested Comments

Gosu supports nested comments. The purpose of nested comments is to quickly comment out large swaths of code temporarily while avoiding compiler errors whenever the enclosed code contains comments.

In earlier releases, the Gosu compiler searched only for “`/*`” after encountering a comment that opened with “`/*`”. This behavior permitted developers to include dividing lines within lengthy comments, like the following example.

```
////////////////////////////////////////////////////////////////////////
```

In BillingCenter 8.0.4, the Gosu compiler searches for “`/*`” after encountering a comment that opens with “`/*`” in case the comment body contains a nested comment. Because the comment line in the preceding example begins with “`/*`”, the compiler begins searching for the close of the nested comment and never finds one.

Following an upgrade to BillingCenter 8.0.4, the Gosu compiler may produce the following error message:

```
unclosed comment
This occurs in multiple-line comments that use the open and close comment marks "/*" and "*/" if the
comment body contains the character sequence "/*".
```

To resolve unclosed comment errors

1. If the Gosu compiler reports the unclosed comment error, open the source file in Studio.
2. Rewrite any comments that inadvertently include the character sequence “/*” within the body of comments. In the preceding example, you could avoid the problem by inserting a space between the slash and the asterisk or by changing to a sequence of characters other than asterisks.
If there are a number of errors for one source file, consider opening the source in the pre-upgrade version of Studio. Then you can compare the commented sections between the old and new Gosu behavior.
3. Compile the project to find any further errors.

Upgrading Rules to BillingCenter 8.0.4

The Configuration Upgrade Tool does not upgrade rules. The tool classifies rules in the unmergeable filter. Within the target directory, Guidewire-provided default rules are located in `modules/configuration/config/rules`. The Configuration Upgrade Tool moves your custom rules to `modules/configuration/config/rules`.

Guidewire also copies the default rules for the current release to a BillingCenter 8.0.4 Rules folder within `modules/configuration/config/rules`. Use Studio to update your rules. You can use the rules in the BillingCenter 8.0.4 folder as a comparison. Compare your custom rules to the new default 8.0.4 versions and update your rules as needed.

You might find it useful to do a bulk comparison of default rules from the base release against the 8.0.4 versions to determine what types of changes Guidewire has made.

To compare rules between versions using the Rule Repository Report

1. If you want to compare default rules only, temporarily remove custom rules from your starting version by moving the `modules/configuration/config/rules` directory to a location outside the BillingCenter directory.
If you want to compare your custom rules against the BillingCenter 8.0.4 rules, do not move the `modules/configuration/config/rules` directory from your starting version. However, do remove the `BillingCenter<base version>` directory from `modules/configuration/config/rules/rules` of the starting version if this directory exists.
2. Open a command window.
3. Navigate to the `bin` directory of your starting version.
4. Enter the following command:
`gwbc regen-rulereport`
This command creates a rule repository report XML file in `build/rules`.
5. Append the starting version number to the XML file name.
6. Restore moved directories to the starting version.
7. Install files for a fresh BillingCenter 8.0.4 version. This is a separate configuration from the target configuration that you have merged. This version will only contain the default rules provided with BillingCenter 8.0.4.
8. Navigate to the `bin` directory of the new BillingCenter 8.0.4 version.
9. Enter the following command:
`gwbc regen-rulereport`

This command creates a rule repository report XML file in `build/rules`. There is a slight change to the path between the versions.

10. Append the target version number to the XML file name.
11. Open both rule report XML files in a merge tool. You do not merge base rules using the rule repository reports. However, looking at changes that Guidewire has made to the base rules can help you determine the types of changes you must make in your custom rules.
In your merge tool, disable whitespace differences and comments to reduce the amount of inconsequential differences shown between rules.

Update custom rules using Studio. Studio does not compare your rules directly with target rules. However, Studio provides powerful Gosu editing features not available in a standard text editor that can alert you to issues.

In Studio, you can compare custom rules to default BillingCenter 8.0.4 rules by opening the default rules in the BillingCenter 8.0.4 directory within `configuration → config → Rule Sets`. When you have finished updating all of your custom rules, delete the BillingCenter 8.0.4 rules directory from `modules/configuration/config/rules`.

The BillingCenter 8.0.4 default rules are enabled because some features depend on these rules.

Running PCF Iterator Upgrade

Prior to BillingCenter 7.0, toolbar buttons, with the exception of `CheckedValues`, did not specify an iterator. BillingCenter used the first iterator defined in the PCF file following the button. In BillingCenter 7.0, all iterator buttons have an `iterator` attribute set. Each toolbar button must have a specified iterator on which it operates.

BillingCenter 7.0 includes a command-line tool to update PCF files to specify an iterator for toolbar buttons. The tool selects the first iterator defined in the PCF file following the button.

The tool only works on PCF files in the `configuration` module. You could have a PCF file that references a list view defined in another PCF file. However, the PCF file that defines the list view is unchanged from the default configuration and is therefore not in the `configuration` module. In this case, copy the PCF file containing the list view definition to the `configuration` module, preserving the directory structure.

To run the PCF iterator upgrade

1. In the target BillingCenter 8.0.4 directory, navigate to the `bin` directory.
2. Run the following command:

```
gwbc iterator-upgrade
```

Translating New Display Properties and Typecodes

BillingCenter 8.0.4 adds new display properties and typecodes. If you have defined additional locales, export these new display properties and typecodes to a file, define localized values, and reimport the localized values. If you do not have additional locales defined in your BillingCenter environment, skip this procedure.

To localize new display properties and typecodes

1. Export display keys by running the following command from your BillingCenter 8.0.4 environment `BillingCenter/bin` directory:

```
gwbc export-l10ns -Dexport.file="translation_file" -Dexport.locale="language to export"
```

2. Open the exported translation file in a text editor. The first section of the file lists display properties and typecodes that have a localized value. The second section lists display properties and typecodes that do not have a localized value.
3. Specify localized values for the untranslated properties.
4. Save the updated file.
5. Import the updated file by running the following command from your BillingCenter 8.0.4 environment `BillingCenter/bin` directory:
`gwbc import-l10ns -Dimport.file="translation_file" -Dimport.locale="language to import"`
After you import the localized typecodes and display keys, you can view them in Studio.

Validating the BillingCenter 8.0.4 Configuration

This topic includes procedures to validate the upgraded configuration.

Using Studio to Verify Files

You can use Studio to verify classes and enhancements, including libraries, PCF files, rules, and typelists without having to start BillingCenter. Do not start BillingCenter at this point. Studio can run without connecting to the application server.

To validate Studio resources

1. Start Guidewire Studio by running `gwbc studio` from the `BillingCenter\bin` directory.
2. Click **Analyze** → **Inspect Code....**
3. Set the **Inspection scope** to module 'configuration'.
4. Click **OK**. Studio runs inspections to identify incorrect Gosu syntax, issuing either a warning or an error.
5. Correct all identified errors with Studio. You can defer fixing warnings.

Starting BillingCenter and Resolving Errors

IMPORTANT In the process described in this section, do not point the BillingCenter server at a production database. The goal of this process is to test the configuration upgrade. Create an empty database account and point BillingCenter to this account for this process. See “Configuring the Database” on page 23 in the *Installation Guide* and “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.

Upon starting the server for the first time, you might receive errors that prevent the server from starting. In general, fixing errors and starting the server is an iterative process that involves:

1. Start the server for the target configuration.
BillingCenter encounters a configuration error and shuts down.
2. Copy the error message to a log file.
3. Locate the configuration causing the error, such as a line of code in a PCF.
4. Comment out the offending line.

After the server starts successfully, look at the log and start solving errors and introducing solutions into the environment. Assign errors to developers on your team.

5. Copy the commented file to the test bed for later analysis.
6. Begin again with step 1. Continue until the server starts successfully.

When the server starts successfully, resolve any remaining issues in the configuration that caused startup errors. Attempt to resolve each error individually and start the server to see if the fix worked.

Building and Deploying BillingCenter 8.0.4

After you apply and validate an upgrade to the configuration environment, rebuild and redeploy BillingCenter. Before you begin, make sure you have carefully prepared for this step. In particular, make sure you have updated your infrastructure appropriately.

Review this topic and then rebuild and redeploy BillingCenter to the application server. See “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide* of the target version for instructions.

WARNING Do not yet start BillingCenter. Only package the application file and deploy it to the application server. Starting BillingCenter begins the database upgrade.

If you have multiple Guidewire products, then upgrade, build, and deploy each individually before attempting to reintegrate them.

The Build Environment

With the exception of the database configuration, the first time you start the application server use the unmodified `config.xml` and `logging.properties` files provided with the target configuration. After the server starts successfully, you can merge in specific configurations of these files.

If you encounter build failures due to data dictionary generation, you can comment out this dictionary generation. Then, as you start the server, it reports any PCF configuration errors. After you have corrected PCF configurations, un-comment the dictionary generation and rebuild the application file.

Preserving JAR Files

Place custom JAR files in the `/config/lib` directory. Building and deploying a WAR or EAR file copies the JAR file into the appropriate place for it to be accessed by BillingCenter. JAR files in this location survive the upgrade process.

Upgrading the BillingCenter 3.0.x Database

This topic provides instructions for upgrading the BillingCenter database to BillingCenter 8.0.4.

If you are upgrading from a 7.0.x version, see “Upgrading the BillingCenter 7.0.x Database” on page 133 instead.

This topic includes:

- “Upgrading Administration Data for Testing” on page 250
- “Identifying Data Model Issues” on page 251
- “Verifying Batch Process and Work Queue Completion” on page 252
- “Purging Data Prior to Upgrade” on page 252
- “Validating the Database Schema” on page 253
- “Checking Database Consistency” on page 254
- “Creating a Data Distribution Report” on page 254
- “Generating Database Statistics” on page 255
- “Creating a Database Backup” on page 256
- “Updating Database Infrastructure” on page 256
- “Preparing the Database for Upgrade” on page 256
- “Setting Linguistic Search Collation” on page 257
- “Field Encryption and the Upgraded Database” on page 258
- “Customizing the Upgrade” on page 258
- “Running the Commission Payable Calculations Process” on page 269
- “Configuring the Database Upgrade” on page 269
- “Checking the Database Before Upgrade” on page 276

- “Disabling the Scheduler” on page 276
- “Suspending Message Destinations” on page 277
- “Starting the Server to Begin Automatic Database Upgrade” on page 277
- “Viewing Detailed Database Upgrade Information” on page 314
- “Dropping Unused Columns on Oracle” on page 315
- “Exporting Administration Data for Testing” on page 315
- “Upgrading Phone Numbers” on page 317
- “Final Steps After The Database Upgrade is Complete” on page 318

Upgrading Administration Data for Testing

You might want to create an upgraded administration data set for development and testing of rules and libraries with BillingCenter 8.0.4. You can wait until the full database upgrade is complete and then export the administration data, as described in “Exporting Administration Data for Testing” on page 315. Or, you can upgrade only the administration data to have this data available earlier in the upgrade process. Use the procedure in this section to create an upgraded administration data set before upgrading the full database.

To upgrade administration data

1. Export administration data from your current (pre-upgrade) BillingCenter production instance:
 - a. Log on to BillingCenter as a user with the `viewadmin` and `soapadmin` permissions.
 - b. Click the **Administration** tab.
 - c. Choose **Import/Export Data**.
 - d. Select the **Export** tab.
 - e. Select **Admin** from the **Data to Export** dropdown.
 - f. Click **Export**. BillingCenter exports an `admin.xml` file.
2. On a new pre-upgrade development environment based on your production configuration, create an empty version of `importfiles.txt` in the `modules/configuration/config/import/gen` directory.
3. Create empty versions of the following CSV files:
 - `activity-patterns.csv`
 - `authority-limits.csv`
 - `reportgroups.csv`
 - `roleprivileges.csv`
 - `rolereportprivileges.csv`Leave `roles.csv` as the original complete file.
4. Start the development environment server by opening a command prompt to `BillingCenter/bin` and entering the following command:
`gwbc dev-start`
5. Import this administration data into the development environment.
 - a. Log on to BillingCenter as a user with the `viewadmin` and `soapadmin` permissions.
 - b. Click the **Administration** tab.
 - c. Choose **Import/Export Data**.
 - d. Select the **Import** tab.

- e. Click **Browse....**
- f. Select the `admin.xml` file that you exported in step 1.
- g. Click **Open**.
6. Create a backup of the new development environment database.
7. Create a new database account for the development environment on a database management system supported by BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <http://guidewire.custhelp.com>.
See “Configuring the Database” on page 23 in the *Installation Guide* for instructions to configure the database account.
8. Restore the backup of the database containing the imported administration data into the new database.
9. Connect your upgraded target BillingCenter 8.0.4 configuration to the restored database.
10. Start the BillingCenter 8.0.4 server to upgrade the database.
11. Export the upgraded administration data:
 - a. Start the BillingCenter 8.0.4 server by navigating to `BillingCenter/bin` and running the following command:
`gwbc dev-start`
 - b. Open a browser to BillingCenter 8.0.4.
 - c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
 - d. Click the **Administration** tab.
 - e. Choose **Import/Export Data**.
 - f. Select the **Export** tab.
 - g. For **Data to Export**, select **Admin**.
 - h. Click **Export**. Your browser will note that you are opening a file and will prompt you to save or download the file.
 - i. Select to download the `admin.xml` file. You can import this XML file into local development environments of BillingCenter 8.0.4.

Identifying Data Model Issues

Before you upgrade a production database, identify issues with the datamodel by running the database upgrade on an empty database. This process does not identify all possible issues. The database upgrade does not detect issues caused by specific data in your production database. Instead, this procedure identifies issues with the data model.

Complete the following procedure to identify data model issues, and correct any issues on an empty schema. Then, follow the full list of procedures in this topic to upgrade a production database. This list begins with “Verifying Batch Process and Work Queue Completion” on page 252 and finishes with “Final Steps After The Database Upgrade is Complete” on page 318.

To identify data model issues

1. Create an empty schema of your starting version database. You can do this in a development environment by pointing the development BillingCenter installation at an empty schema and starting the BillingCenter server. See “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.
2. Complete the configuration upgrade for data model files in your starting version, according to the instructions in “Upgrading the BillingCenter 3.0.x Configuration” on page 203. You do not need to complete the merge process for all files.
3. Configure your upgraded development environment to point to the database account containing the empty schema of your old version. See “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.
4. Start the BillingCenter server in your upgraded development environment. The server performs the database upgrade to BillingCenter 8.0.4. See “Starting the Server to Begin Automatic Database Upgrade” on page 277.
5. Check for errors reported during the upgrade process. Resolve any issues before upgrading your production database. You can use the `IDatabaseUpgrade` plugin to run custom SQL before and after the database upgrade. For more information, see “Running Custom Version Checks and Triggers” on page 259.

Verifying Batch Process and Work Queue Completion

All batch processes and work queues must complete before beginning the upgrade. Check the status of batch processes and work queues in your current production environment.

To check the status of batch processes and work queues

1. Log in to BillingCenter as the superuser.
2. Press Alt + Shift + T. BillingCenter displays the **Server Tools** tab.
3. Click **Batch Process Info**.
4. Select **Any** from the **Processes** drop-down filter.
5. Click **Refresh**.
6. Check the **Status** column for each batch process listed. This list also includes batch processes that are writers for distributed work queues. If any of the batch processes have a **Status** of **Active**, wait for the batch process to complete before continuing with the upgrade.

Purging Data Prior to Upgrade

This topic includes recommendations for purging certain types of data from the database prior to upgrade. Removing unused records can improve the performance of the database upgrade and BillingCenter.

Purging Old Messages from the Database

Purge completed inactive messages before upgrading the database. Doing so reduces the complexity of the database upgrade.

Use the following command from the current (pre-upgrade) customer configuration `admin/bin` directory to purge completed messages from the `bc_MessageHistory` table:

```
messaging_tools -password password -server http://server:port/instance -purge MM/DD/YYYY
```

This tool deletes completed messages with a send time before the date `MM/DD/YYYY`.

Or, you can use the following web service API:

```
IMessageToolsAPI.purgeCompletedMessages(java.util.Calendar cutoff)
```

Periodically purge old messages to prevent the database from growing unnecessarily.

Purge messages from the database before starting BillingCenter, so the database upgrade does not attempt to convert those rows.

You cannot resend old messages after the upgrade. This is because integrations change and the message payload might be different. It is important that messages that have failed or not yet been consumed finish prior to upgrading.

After you purge completed inactive messages, reorganize the `bc_MessageHistory` table. You might also want to rebuild any indexes on the table. Contact Guidewire Support if you need assistance.

Purging Completed Workflows and Workflow Logs

Each time BillingCenter creates an activity, the activity is added to the `bc_Workflow`, `bc_WorkflowLog` and `bc_WorkflowWorkItem` tables. Once a user completes the activity, BillingCenter sets the workflow status to completed. The `bc_Workflow`, `bc_WorkflowLog` and `bc_WorkflowWorkItem` table entry for the activity are never used again. These tables grow in size over time and can adversely affect performance as well as waste disk space. Excessive records in these tables also negatively impacts the performance of the database upgrade.

Remove workflows, workflow log entries, and workflow items for completed activities to improve database upgrade and operational performance and to recover disk space.

BillingCenter includes work queues to purge completed workflows and their logs that are older than a configurable number of days. Guidewire recommends that you purge completed workflows and their logs periodically. This reduces performance issues caused by having a large number of unused workflow log records.

To set the number of days after which the `purgeworkflows` process purges completed workflows and their logs, set the following parameter in `config.xml`:

```
<param name="WorkflowPurgeDaysOld" value="value" />
```

Set the value to an integer. By default, `WorkflowPurgeDaysOld` is set to 60. This is the number of days since the last update to the workflow, which is the completed date.

You can launch the Purge Workflows batch process from the `BillingCenter/admin/bin` directory with the following command:

```
maintenance_tools -password password -startprocess PurgeWorkflows
```

You can also purge only the logs associated with completed workflows older than a certain number of days. Run the `purgeworkflowlogs` process instead. This process leaves the workflow records and removes only the workflow log records. The `purgeworkflowlogs` process is configured using the `WorkflowLogPurgeDaysOld` parameter rather than `WorkflowPurgeDaysOld`.

You can launch the Purge Workflow Logs batch process from the `BillingCenter/admin/bin` directory with the following command:

```
maintenance_tools -password password -startprocess PurgeWorkflowLogs
```

Validating the Database Schema

This validation detects the unlikely event that the data model defined by your configuration files has become out of sync with the database schema. While the pre-upgrade server is running, use the `system_tools` command in `admin/bin` of the customer configuration to verify the database schema:

```
system_tools -password password -verifydbschema -server servername:port/instance
```

Correct any validation problems in the database before proceeding. Contact Guidewire Support for assistance.

Following the database upgrade, run this command again from the `admin/bin` directory of the target (upgraded) configuration.

Checking Database Consistency

BillingCenter has hundreds of internal database consistency checks. Before upgrading, run consistency checks to verify the integrity of your data.

Run database consistency checks early in the upgrade project. Fix any consistency errors. Continue to periodically run consistency checks and resolve issues so that your database is ready to upgrade when you begin the upgrade procedure. Consistency issues might take some time to resolve, so begin the process of running consistency checks and fixing issues early. Contact Guidewire Support for information on how to resolve any consistency issues.

After the database upgrade, run consistency checks again from the BillingCenter **Consistency Checks** page.

To run consistency checks

1. Start the BillingCenter server if it is not already running.
2. Log in to BillingCenter with an administrator account.
3. Press Alt + Shift + T to access the **Server Tools**.
4. Click **Info Pages**.
5. Select **Consistency Checks** from the drop-down list.
6. To increase the number of threads used to run consistency checks, increase the **Number of threads**. The number of threads to use depends on the capability of your database server. Increasing the number of threads can improve performance of consistency checks as long as your server can process the threads. Guidewire recommends starting with five threads. If too many threads are used, there is a greater chance that current users experience reduced performance if the database server is fully loaded.

To set the number of threads in versions prior to 8.0, specify a value for the `checker.threads` parameter within the database block of `config.xml`.

```
<database>
  ...
  <param name="checker.threads" value="5" />
  ...
</database>
```

7. Click **Run Consistency Checks**.

See also

“[Checking Database Consistency](#)” on page 36 in the *System Administration Guide*.

Creating a Data Distribution Report

Generate a data distribution report for the database before an upgrade. Save the output of this report. Run the report again after the upgrade to ensure the distribution is still correct.

Guidewire is very interested in the data distribution of your databases. Guidewire uses these reports to better understand the nature of your database and to optimize BillingCenter performance. Guidewire appreciate copies of your reports, both before and after upgrades.

You can also use this information to tune the application server cache. See “Application Server Caching” on page 63 in the *System Administration Guide*.

To create a database distribution report

1. In config.xml, set <param name="EnableInternalDebugTools" value="true"/>.
2. Start the BillingCenter application server.
3. Log into BillingCenter as an administrative user.
4. Type ALT + SHIFT + T while in any screen to reach the **Server Tools** page.
5. Choose **Info Pages** from the **Server Tools** tab.
6. Choose the **Data Distribution** page from the **Info Pages** dropdown.
7. Enter a reason for running the Data Distribution batch job in the **Description** field.
8. On this page, select the **Collect distributions for all tables** radio button and check all checkboxes to collect all distributions.
9. Push the **Submit Data Distribution Batch Job** button on this page to start the data collection.
10. Return to the **Data Distribution** page and push its **Refresh** button to see a list of all available reports. The batch job has completed when the **Available Data Distribution** list on the **Data Distribution** page includes your description.
11. Select the desired report and use the **Download** button to save it zipped to a text file. Unzip the file to view it.

Generating Database Statistics

To optimize the performance of the BillingCenter database, it is a good idea to update database statistics on a regular basis. Both SQL Server and Oracle can use these statistics to optimize database queries.

If you update database statistics on a regular basis, you do not need to update statistics before an upgrade. If you do not update database statistics on a regular basis, Guidewire recommends that you update incremental statistics before running the upgrade.

To generate incremental database statistics

1. Get the proper SQL statements for updating the statistics in BillingCenter tables by running the following command in the pre-upgrade environment:

```
maintenance_tools -getincrementaldbstatisticsstatements -password password  
-server http://server:port/instance > db_stats.sql
```

2. Run the resulting SQL statements against the BillingCenter database.

You can configure SQL Server to periodically update statistics. See your database documentation and “Configuring Database Statistics” on page 39 in the *System Administration Guide* for more information.

The database upgrade can take a long time, and has built-in statistics collection that help you see if any part of the upgrade is slow. Collect these statistics, and compare them to the statistics you collected before the upgrade. The config.xml file has parameters that control this statistics collection.

If you disabled statistics collection during the upgrade by setting updatestatistics to false, Guidewire recommends that you generate full statistics as soon as possible after the upgrade. For instructions, see “Commands for Updating Database Statistics” on page 40 in the *System Administration Guide*. Note that the commands for generating statistics have moved to system_tools instead of maintenance_tools in the upgraded BillingCenter.

Creating a Database Backup

Prepare the environment so that you can make a total recovery of the original installation if you run into problems during the upgrade.

The first time you start the BillingCenter server after running the upgrade tool, the server updates the database. During its work, the database upgrader minimizes the logging that it does. For these reasons, back up your database before starting an upgrade. Your pre-upgrade database might not be recoverable after an upgrade.

Updating Database Infrastructure

Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

Preparing the Database for Upgrade

This topic notes steps to prepare the database for the upgrade process.

Ensuring Adequate Free Space

The database upgrade requires significant free space. Make sure the database has at least 50% of the current database size available as free space.

Disabling Replication

Disable database replication during the database upgrade.

Assigning Default Tablespace (Oracle only)

Set the default tablespace for the database user to the one mapped to the logical tablespace OP in config.xml.

The database upgrade creates temporary tables during the upgrade without specifying the tablespace. If the Oracle database user was created without a default tablespace, Oracle by default creates the tables in the SYSTEM tablespace. The Guidewire database user is likely not to have the required quota permission on the SYSTEM tablespace. This results in an error of the type:

```
java.sql.SQLException: ORA-01950: no privileges on tablespace 'SYSTEM'
```

Even if the default tablespace is not SYSTEM, if the Guidewire database user does not have quota permission on the default tablespace, the temporary table creation during upgrade fails.

Setting Linguistic Search Collation

WARNING For SQL Server, compare the default collation of the database to the collation defined for your locale. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your SQL Server database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, this adds time to the total database upgrade process.

WARNING Oracle Java Virtual Machine (JVM) must be installed on all Oracle databases hosting BillingCenter. The only exception is when the BillingCenter application locale is English and you only require case-insensitive searches. Ensure that Oracle initialization parameter `java_pool_size` is set to a value of above 50 MB.

You can specify how you want BillingCenter to collate search results. The `strength` attribute of the `LinguisticSearchCollation` element of `GWLocale` for the default locale in `localization.xml` specifies how BillingCenter sorts search results. You can set the `strength` to `primary` or `secondary`.

With `LinguisticSearchCollation strength` set to `primary`, BillingCenter searches results in a case-insensitive and accent-insensitive manner. BillingCenter considers an accented character equal to the unaccented version of the character if the `LinguisticSearchStrength` for the default application locale is set to `primary`. For example, with `LinguisticSearchCollation strength` set to `primary`, BillingCenter treats “Renée”, “Renee”, “renee” and “reneé” the same.

With `LinguisticSearchCollation strength` set to `secondary`, BillingCenter searches results in a case-insensitive, accent-sensitive manner. BillingCenter does not consider an accented character equal to the unaccented version of the character if the `LinguisticSearchCollation strength` for the default application locale is set to `secondary`. For example, with `LinguisticSearchCollation strength` set to `secondary`, a BillingCenter search treats “Renee” and “renee” the same but treats “Renée” and “reneé” differently. By default, BillingCenter uses a `LinguisticSearchCollation strength` of `secondary`, for case-insensitive, accent-sensitive searching.

The `collations.xml` file defines the collations to use for different locales and different collation strengths. The `primary`, `secondary`, and `tertiary` attributes of the `Collation` element define the collation to use depending on the `LinguisticSearchCollation strength` attribute in `localization.xml`.

BillingCenter 7.0 introduced configurable linguistic searching for SQL Server databases. In releases prior to BillingCenter 7.0, BillingCenter used the collation setting of the database server. If you are satisfied with the existing linguistic searching mechanism, check that the collation of your database matches the collation defined in `collations.xml` for the locale and strength. If the collations do not match, then the database upgrade changes the collation attribute for all denormalized columns created for searching. This results in dropping and recreating any dependent indexes on these columns. Depending on the size of these tables, this adds time to the total database upgrade process.

For sorting search results, the following rules apply:

- **Case** – All searches ignore the case of the letters, whether `LinguisticSearchCollation strength` is set to `primary` or `secondary`. “McGrath” equals “mcgrath”.
- **Punctuation** – Punctuation is always respected, and never ignored. “O'Reilly” does not equal “OReilly”.
- **Spaces** – Spaces are respected. “Hui Ping” does not equal “HuiPing”.
- **Accents** – An accented character is considered equal to the unaccented version of the character if `LinguisticSearchCollation strength` is set to `primary`. An accented character is not equal to the unaccented version if `LinguisticSearchCollation strength` is set to `secondary`.

Japanese only

- **Half Width/Full Width** – Searches under a Japanese locale always ignore this difference.
- **Small/Large Kana** – Japanese small/large letter differences are ignored only when `LinguisticSearchCollation strength` is set to `primary`, meaning accent-insensitive.
- **Katakana/Hiragana sensitivity** – Searches under a Japanese locale always ignore this difference.
- The long dash character is always ignored.
- Soundmarks (` and °) are only ignored if `LinguisticSearchCollation strength` is set to `primary`.

German only

- Vowels with an umlaut compare equally to the same vowel followed by the letter e. Explicitly, “ä”, “ö”, “ü” are treated as equal to “ae”, “oe” and “ue”.
- The Eszett, or sharp-s, character “ß” is treated as equal to “ss”.

BillingCenter populates denormalized values of searchable columns to support the search collation. For example, with `LinguisticSearchCollation strength` set to `primary`, BillingCenter stores the value “Renée”, “Renée”, “renée” and “reneé” in a denormalized column as “renée”. With `LinguisticSearchCollation strength` set to `secondary`, BillingCenter stores a denormalized value of “renée” for “Renée” or “renée” and stores “reneé” for “Renée” or “reneé”. Japanese and German locales make additional changes when storing values in denormalized columns in order to conform to the rules listed previously for those locales.

Any time you change the `LinguisticSearchCollation strength` and restart the server, BillingCenter repopulates the denormalized columns. Previous versions of BillingCenter populated the denormalized columns with lowercase values for case-insensitive search, equivalent to setting `LinguisticSearchCollation strength` to `secondary`. If you set `LinguisticSearchCollation strength` to `primary`, BillingCenter repopulates the denormalized columns, substituting any accented characters for their base equivalents. This process can take a long time, depending on the amount of data. Therefore, if you want to change `LinguisticSearchCollation strength` to `primary`, you might want to do so after the database upgrade. If you are concerned about the duration of the database upgrade, you can change your search collation settings after the upgrade. During a maintenance period, change `LinguisticSearchCollation strength` to `primary` and restart the server to repopulate the denormalized columns.

For Japanese locales, the BillingCenter database upgrade from a prior major version repopulates the denormalized columns regardless of the `LinguisticSearchCollation strength` value. BillingCenter must repopulate the denormalized columns for Japanese locales to have search results obey the Japanese-only rules listed previously.

Field Encryption and the Upgraded Database

The database upgrader handles most encrypted fields with no problem. If you have started to use field encryption and have changed fields in the database from no field encryption to encrypted, specifying some encryption algorithm, the upgrader preserves this encryption. However, if you later change the encryption to another algorithm type, the upgrader does not handle this case. See “[Changing Your Encryption Algorithm Later](#)” on page 258 in the *Integration Guide*.

Customizing the Upgrade

The `IDatamodelUpgrade` plugin interface provides hooks for custom code that you want to run during the database upgrade. You can use the `IDatamodelUpgrade` plugin to:

- execute custom version checks to test data or the data model itself before starting the upgrade.
- make custom database changes before or after the database upgrade.
- make data model changes to archived entities.

For example, you might fix a consistency check failure issue, correct issues reported by version checks, or delete a custom extension that you are no longer using.

IMPORTANT BillingCenter 3.0 included a similar plugin interface, `IDatabaseUpgrade`. If you previously implemented `IDatabaseUpgrade` for an upgrade to BillingCenter 3.0, you must now implement `IDatamodelUpgrade` if you want to execute custom upgrade code.

Running Custom Version Checks and Triggers

You can use the `IDatamodelUpgrade` plugin to run custom version checks and triggers before and after the database upgrade. The `IDatamodelUpgrade` plugin interface contains method signatures for two methods that you must define in your plugin. These signatures are:

- `property getBeforeUpgradeDatamodelChanges() : List<IDataModelChange<BeforeUpgradeVersionTrigger>>`
- `property getAfterUpgradeDatamodelChanges() : List<IDataModelChange<AfterUpgradeVersionTrigger>>`

Each method returns a list of `IDataModelChange` entities, each taking a `BeforeUpgradeVersionTrigger` or `AfterUpgradeVersionTrigger` type parameter. The `IDataModelChange` interface has two methods that you use to make data model changes. The `getDatabaseUpgradeVersionTrigger` method is for changes to the database. The `getArchivedDocumentUpgradeVersionTrigger` method is for changes to archived entities. If your organization has not implemented archiving or you do not want to make changes to archived entities, return null for `getArchivedDocumentUpgradeVersionTrigger`.

The `getAfterUpgradeDatamodelChanges` method runs after the Guidewire upgrade version triggers. You can use this method to move data into extension tables or columns that did not exist prior to upgrading.

You can return an empty list from either `getBeforeUpgradeDatamodelChanges` or `getAfterUpgradeDatamodelChanges`. For example, if you only have triggers to run before the upgrade, you can return an empty list from `getAfterUpgradeDatamodelChanges`.

Modifying Tables

Both `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` base classes provide a protected `getTable` method that accepts a `string` parameter. The `getTable` method returns an `IBeforeUpgradeTable` or `IAfterUpgradeTable` object that provides a number of methods for DDL and DML operations, such as:

- `create` – Create the table if it does not already exist. The table must be related to an entity defined in the data model. This method is available only for `IBeforeUpgradeTable`.
- `delete` - Deletes rows from a table. Returns a builder (`IBeforeUpgradeDeleteBuilder` for `IBeforeUpgradeTable`, `IDeleteBuilder` for `IAfterUpgradeTable`) that has methods for comparing data to restrict which rows are deleted.
- `drop` - Drops the table.
- `dropColumns` - Drops multiple columns from the table.
- `getColumn` – Returns an `IBeforeUpgradeColumn` or `IAfterUpgradeColumn` object that has methods to perform DDL operations on the column such as create, drop, rename, and more.
- `insert` – Returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable`) to perform an insert operation.
- `insertSelect` – Returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`) for SQL to perform an insert operation using data selected from a table.
- `rename` - Renames the table.
- `update` – Returns a builder (`IBeforeUpgradeUpdateBuilder` for `IBeforeUpgradeTable`, `IUpdateBuilder` for `IAfterUpgradeTable`) for SQL to perform an update operation.

For DML operations, call the `execute` method on the builder to actually perform the operation. The `execute` method runs in its own transaction. You do not need to handle transactions and `TransactionManager`.

There are more methods on the `IBeforeUpgradeTable` and `IAfterUpgradeTable` classes documented in the Guidewire Gosu API documentation. To generate the Guidewire Gosu API documentation, run the `gwbc regen-gosudoc` command from the BillingCenter `bin` directory. Then, open `BillingCenter/build/gosudoc/index.html`.

The methods for `BeforeUpgradeVersionTrigger` intentionally take strings but not entities or properties. This is because the name of the column could change in the future. Consider `PropertyA` on `EntityE` which corresponds to column A in the database. Suppose you use `PROPERTYA_PROP` in a version trigger at minor version 200, but at minor version 250, you decide to rename the backing column from A to B. The version trigger you wrote in the past would break because it would execute before the rename operation and would try to use the new column name.

`AfterUpgradeVersionTrigger` is very similar to `BeforeUpgradeVersionTrigger`. A few differences include:

- The `AfterUpgradeVersionTrigger` DML builders use the query builder, `IQueryBuilder`.
- In an `AfterUpgradeVersionTrigger` you can use properties and types in addition to strings.
- Some DDL operations are not provided on the `IAfterUpgradeTable` object, including creating a table or adding a column.

Unless you require one of the unique capabilities of `AfterUpgradeVersionTrigger`, use `BeforeUpgradeVersionTrigger` for custom version triggers.

Upgrading Typelists

The `BeforeUpgradeVersionTrigger` class includes a `getTypeKeyID` method with the following signature:

```
protected final Integer getTypeKeyID(IEntityType subtype)
```

Note: Protected methods do not appear in the Gosu documentation. Use CTRL + SPACE in Studio to show available methods and properties.

The `getTypeKeyID` method returns the integer ID of the type code in the type list matching the given table name. This method checks both the existing typelist tables and the metadata files to determine what all typekey IDs will be after upgrade. Therefore, the `getTypeKeyID` method works as expected even before a new typekey or typelist table is created during the automatic schema upgrade phase.

This method also works for orphaned typecodes that have not yet been removed from the database. These are typecodes that still exist in the database table but not in the metadata file. You can use the `getTypeKeyID` method for remapping usages of orphaned typecodes.

Version Checks

In some cases, you might want to check for a certain condition in the database before the upgrade proceeds. This is referred to as a version check. Only read operations are available in version checks. For example, you can write a version check to query a table or check the existence of a table or column, but the check cannot insert new rows. The `BeforeUpgradeVersionTrigger` class includes a `hasVersionCheck` method that you must define to return true or false. If the trigger does include a version check, overwrite the `createVersionCheck` method to define your custom version check. For standalone version checks that are not associated with a version trigger, you can use `BeforeUpgradeVersionCheckWrapper`.

The upgrade executes all custom version checks before custom version triggers. The upgrade runs Guidewire version checks after all custom `BeforeUpgradeVersionTrigger` implementations, so you can create a `BeforeUpgradeVersionTrigger` to correct issues detected by the Guidewire version checks.

If a custom version check fails, the upgrade stops before running any upgrade triggers. Correct the issue and restart the upgrade.

Order of Execution

The upgrade performs actions in the following order:

Step	Action	In the event of failure due to a data issue...
1	Custom version checks	Correct the data issue. Restart the upgrade. You do not need to restore the database because the upgrade has not made any changes.
2	Custom BeforeUpgradeVersionTrigger implementations	Restore the database from a backup. Correct the data issue. Consider adding custom version checks to test for other instances of the data issue.
3	Guidewire version checks	If you do not have any custom BeforeUpgradeVersionTrigger implementations, correct the data issue and restart the upgrade. If you do have custom BeforeUpgradeVersionTrigger implementations, restore the database from a backup. Then, correct the data issue. In either case, consider creating a custom BeforeUpgradeVersionTrigger implementation to correct the data issue.
4	Guidewire version triggers	A failure due to data issues at this stage is unlikely. Contact Guidewire Support.
5	Automated data model upgrade to update the database to the defined data model	A failure due to data issues at this stage is unlikely. Contact Guidewire Support.
6	Guidewire version triggers that require the updated data model in the database	A failure due to data issues at this stage is unlikely. Contact Guidewire Support.
7	Custom AfterUpgradeVersionTrigger implementations	Restore the database from a backup. Correct the data issue. Consider creating a custom BeforeUpgradeVersionTrigger implementation to correct the data issue if possible.

The preceding table describes failure cases that are caused by data issues. If the upgrade fails for other reasons, such as a disruption of the database server, fix the issue causing the disruption, restore the database, and restart the upgrade.

Versioning

Each `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` instance requires a minor version number, passed as an integer. If the data model version number is less than or equal to the number passed to the instance, then the trigger executes. Whenever you make a data model change, or you want to force an upgrade, increment the version number in `extensions.properties`.

To run custom version checks and triggers

1. Create a new package, such as `companyName.upgrade`, to store your custom version triggers.
 - a. Open Studio.
 - b. In the Studio Project window, expand configuration.
 - c. Right-click `gsrc` and click **New → Package**.
 - d. Enter a package name for upgrade purposes, such as `companyName.upgrade`.
2. Right-click the upgrade package and click **New → Gosu Class**.
3. Enter a name for the class and click **OK**.

- 4.** Create a new Gosu class that extends `CustomerDatamodelUpgrade` and implements `IDatamodelUpgrade`. The class you create must define the `getBeforeUpgradeDatamodelChanges` and `getAfterUpgradeDatamodelChanges` methods. This class is the container from which you call custom version trigger classes.

For example:

```
package companyName.upgrade
uses gw.plugin.upgrade.IDatamodelUpgrade
uses java.lang.Iterable
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger
uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger
uses java.util.ArrayList
uses gw.api.datamodel.upgrade.CustomerDatamodelUpgrade
uses gw.api.datamodel.upgrade.IDatamodelChange
uses gw.api.database.upgrade.DatamodelChangeWithoutArchivedDocumentChange

class TestDatamodelUpgradeImpl extends CustomerDatamodelUpgrade implements IDatamodelUpgrade {

    override property get BeforeUpgradeDatamodelChanges() :
        List<IDatamodelChange<BeforeUpgradeVersionTrigger>> {
            var list = new ArrayList<IDatamodelChange<BeforeUpgradeVersionTrigger>>()
            list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger1()))
            list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new BeforeVersionTrigger2()))
            return list
        }

    override property get AfterUpgradeDatamodelChanges() :
        List<IDatamodelChange<AfterUpgradeVersionTrigger>> {
            var list = new ArrayList<IDatamodelChange<AfterUpgradeVersionTrigger>>()
            list.add(DatamodelChangeWithoutArchivedDocumentChange.make(new AfterVersionTrigger1()))
            return list
        }
}
```

- 5.** Create your custom `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger` Gosu classes. See “`IDatamodelUpgrade` API Examples” on page 262.

- 6.** Implement the `IDatamodelUpgrade` plugin with the new class.

- a. Start Guidewire Studio 8.0.4 by entering `gwbc studio` from the `BillingCenter/bin` directory.
- b. In Studio, expand `configuration` → `config` → `Plugins`.
- c. Right-click `registry` and click `New` → `Plugin`.
- d. In the `Plugin` dialog, enter the name `IDatamodelUpgrade`. For this plugin, the name must match the interface.
- e. In the `Plugin` dialog, click the ... button.
- f. In the `Select Plugin Class` dialog, type `IDatamodelUpgrade` and select the `IDatamodelUpgrade` interface.
- g. In the `Plugin` dialog, click `OK`. Studio creates a `GWP` file under `Plugins` → `registry` with the name you entered.
- h. Click the `Add Plugin` icon (a plus sign) and select `Add Gosu Plugin`.
- i. For `Gosu Class`, enter your class, including the package.
- j. Save your changes.

When you start the server to perform the database upgrade from a prior major version, the upgrade calls the plugin and runs your custom methods.

`IDatamodelUpgrade` API Examples

This topic first introduces the basic structure of a `BeforeUpgradeVersionTrigger` and `AfterUpgradeVersionTrigger`. Next it shows methods that can be included within the `execute` method of these triggers to modify the database.

This topic includes:

- “BeforeUpgradeVersionTrigger Structure” on page 263
- “AfterUpgradeVersionTrigger Structure” on page 264
- “Altering Columns to Match Data Model” on page 264
- “Altering a Non-nullable Column to Nullable” on page 264
- “Creating Columns” on page 265
- “Dropping Columns” on page 266
- “Renaming Columns” on page 266
- “Setting a Column Value for a Specific Subtype” on page 266
- “Creating Tables” on page 267
- “Renaming Tables” on page 267
- “Deleting Rows” on page 267
- “Inserting Rows” on page 268
- “Inserting Data Selected from Another Table” on page 268
- “Updating Rows” on page 269

BeforeUpgradeVersionTrigger Structure

A custom BeforeUpgradeVersionTrigger subclass has the following structure.

```
package companyName.upgrade.before

uses gw.api.database.upgrade.before.BeforeUpgradeVersionCheck
uses gw.api.database.upgrade.before.BeforeUpgradeVersionTrigger

class myBeforeUpgradeTrigger extends BeforeUpgradeVersionTrigger {

    construct() {
        super(dataModelVersionNumber)
    }

    override function execute() {
        // Perform actions here.
    }

    override function hasVersionCheck() : boolean {
        // return true if creating a version check to determine whether the trigger can run.
        // return false if you are not implementing a version check.
    }

    override property get Description() : String {
        return "Description of the version trigger."
    }

    // Override the createVersionCheck method if you are implementing a version check.
    override function createVersionCheck() : BeforeUpgradeVersionCheck {
        return new BeforeUpgradeVersionCheck(dataModelVersionNumber) {

            override function verifyUpgradability() {
                if (condition to detect) {
                    addVersionCheckProblem("description of issue")
                }
            }

            override property get Description() : String {
                return "Description of the version check."
            }
        }
    }
}
```

Define the execute method to perform the actions you want your custom trigger to perform. Some examples are provided in subsequent topics.

AfterUpgradeVersionTrigger Structure

A custom AfterUpgradeVersionTrigger subclass has the following structure.

```
package companyName.upgrade.after

uses gw.api.database.upgrade.after.AfterUpgradeVersionTrigger

class myAfterUpgradeTrigger extends AfterUpgradeVersionTrigger{

    construct() {
        super(dataModelVersionNumber)
    }

    override function execute() {
        // Perform actions here.
    }

    override property get Description(): String {
        return "Description of the version trigger."
    }
}
```

Altering Columns to Match Data Model

In most cases, you do not need to alter a column to match a change to the column type in the logical data model. The upgrader automatically applies data model changes to the database. However, this occurs after all custom BeforeUpgradeVersionTrigger instances have run, so Guidewire provides methods to alter database columns to match the data model.

Altering a single column

If you need to alter a single column for use in a BeforeUpgradeVersionTrigger, modify the data model file, then use the alterColumnTypeToMatchDatamodel method of IBeforeUpgradeColumn.

For example:

```
var table = getTable("TableName")
var column = table.getColumn("ColumnName")
column.alterColumnTypeToMatchDatamodel()
```

Altering multiple column

To alter multiple columns for use in a BeforeUpgradeVersionTrigger, modify the data model file, then use the alterMultipleColumnsToMatchDatamodel method of IBeforeUpgradeTable.

For example:

```
var table = getTable("TableName")
var columnsToChange = new IBeforeUpgradeColumn[2]

columnsToChange[0] = table.getColumn("column1")
columnsToChange[1] = table.getColumn("column1")

table.alterMultipleColumnsToMatchDatamodel(columnsToChange)
```

Altering a Non-nullable Column to Nullable

To alter a column from non-nullable to nullable, use the IBeforeUpgradeColumn method alterColumnTypeToNullable.

For example:

```
var table = getTable("TableName");
table.getColumn("ColumnName").alterColumnTypeToNullable();
```

Creating Columns

The database upgrader automatically creates a column that is added to the data model if the column meets one of the following criteria:

- Nullable
- Non-nullable with a default value specified in the metadata
- Non-nullable without a default value if there are no rows in the table
- The column is an editable field

However, you might want to explicitly create the column in your upgrade trigger if you want the trigger to perform an action on the column such as populating it.

In the data model, the column must be defined as a property on an entity. The database upgrade will determine the correct datatype and nullability from the data model.

Creating a new column is moderately expensive in terms of performance of the upgrade.

Creating a Column

To create a column, invoke the `create` method on the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")  
  
// Create column with given name.  
// Column must be backed by a property on an entity.  
// Upgrader will figure out the correct datatype and nullability.  
  
table.getColumn("ColumnName").create()
```

Creating a Non-nullable Column with an Initial Value

The upgrader throws an exception if you try to add a new non-nullable column without a default value and there are rows in the table. For non-nullable columns, either specify a default value, or create a version trigger that will populate the column.

To create a new column as non-nullable with an initial value, use the `createNonNullableWithInitialValue()` method. In the data model, the column must be defined as non-nullable.

For example:

```
IBeforeUpgradeTable table = getTable("TableName")  
table.getColumn("ColumnName").createNonNullableWithInitialValue(Initial value)
```

The initial value must be of the appropriate type for the column's datatype. You can alter this value in later steps as needed.

Creating a Temporary Column

Use the `createTempColumn` method of `IBeforeUpgradeTable` to add a temporary column to the table. The `createTempColumn` method takes two parameters, a `String` for the column name and an `IDataType` for the column data type. `createTempColumn` creates a new nullable column with the given name and datatype to hold temporary data. You must explicitly drop the temporary column during the upgrade. The schema verifier will report an error during server startup if the column has not been dropped. You can create the temporary column in a `BeforeUpgradeVersionTrigger` and drop it in an `AfterUpgradeVersionTrigger`. This approach is useful when you want to move data from a column that will be removed during the upgrade to a column that will be created during the upgrade.

In the following example, a `BeforeUpgradeVersionTrigger` adds a temporary `shorttext` column to an existing entity and populates it with data from another column on a different entity. An `AfterUpgradeVersionTrigger` moves the data to a new entity.

BeforeUpgradeVersionTrigger Execute Method

```
// Add a temporary column to TableA.
var tableA = getTable("TableA")
var tempColumn = tableA.createTempColumn("tmp_column", DataTypes.shorttext())

// Get an IBeforeUpgradeUpdateBuilder for TableA.
var ub = tableA.update()

// Set the value of the temporary column to the value of ColumnA.
ub.set(tempColumn, ub.getColumnRef("ColumnA"))

ub.execute()
```

AfterUpgradeVersionTrigger Execute Method

```
// Get an IUpdateBuilder for TableA.
var ub = getTable("TableA").update().withLogSQL(true)

var q = new Query(Account).withLogSQL(true)
q.compare("ID", Equals, ub.getQuery().getColumnRef("Account"))
var piDesc = PaymentInstrument.Type.TypeInfo.getProperty("Description") as IEntityPropertyInfo

ub.set(piDesc, q, q.getColumnRef(DBFunction.Expr({"tmp_xyz"}))) // tmp_xyz is the DB table column name
ub.execute()

var tempColumn = getTable("someTable").getColumn("tmp_xyz").drop()
```

Dropping Columns

The upgrader does not drop existing columns in order to prevent data loss. You can write a version trigger to move the data (not shown in example) and then drop the column by using the `drop()` method of the `IBeforeUpgradeColumn`.

For example:

```
var table = getTable("TableName")
table.getColumn("ColumnName").drop()
```

There is a `dropColumns` method on `IBeforeUpgradeTable` to drop multiple columns in one statement. The `dropColumns` method takes an array of `IBeforeUpgradeColumn` objects.

For example:

```
var table = getTable("TableName")
table.dropColumns(table.getColumn("ColumnName2"), table.getColumn("ColumnName3"));
```

In Oracle, dropping a column usually has little effect on upgrade performance. Dropping a column actually marks the column as unused in the metadata. At a later point, the DBA is responsible for performing the necessary cleanup. You can override this functionality and force columns to be dropped right away.

In SQL Server, dropping a column is performance-intensive because the RDBMS has to do some clean up work.

Renaming Columns

To rename a column use the `rename` function on the column object.

```
override function execute() {
    getTable("TableName").getColumn("ColumnName").rename("NewColumnName")
}
```

Setting a Column Value for a Specific Subtype

To set a column to a specific value for specific subtypes, use the `set` and `compare` methods of an `IBeforeUpgradeTable`. Get the typekey ID for comparison using the `BeforeUpgradeVersionTrigger` method `getTypekeyID`.

```
final var myTable = getTable("tableName")
final var myTypecode = getTypeKeyID("typelist name", "typelist code")

final var updateBuilder = myTable.update()

updateBuilder
    .set("myColumn", "some value")
    .compare("subtype", Equals, myTypecode)
```

```
updateBuilder.execute()
```

Creating Tables

To add a new table to the database, define a new entity in the data model. The upgrade creates the table automatically. However, you might want to explicitly create the table in your upgrade trigger if you want the trigger to perform an action on the table such as populating it.

Creating a new table has negligible impact on upgrade performance.

You can create a regular table using the `create` method of `IBeforeUpgradeTable`. The table must first be defined in the data model.

For example:

```
var table = getTable("TableName").create()
```

Creating Temporary Tables

You can add a temporary table to the database based on either the current database schema for a table or the data model definition of a table. You can also create a temporary table with a custom definition.

To create a temporary table based on the current table schema in the database, use the `createNewTempTableBasedOnCurrentSchema` method of `IBeforeUpgradeTable`. The table must be associated with an entity and exist in the database. The returned temporary table will contain the columns that this table has in the database currently. The columns may not match those specified in the entity metadata. For example, the metadata might contain a new column that has not yet been created. The `createNewTempTableBasedOnCurrentSchema` method is usually more appropriate than `createNewTempTableBasedOnThis` if you want to copy data from this table into the new temporary table as the columns will match exactly.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnCurrentSchema()
```

To create a temporary table based on the entity definition of a table in the data model, use the `createNewTempTableBasedOnThis` method of `IBeforeUpgradeTable`. Columns that do not exist in the table are not created on the temporary table, even if the metadata defines such a column. This table may not contain columns that are going to be renamed. The metadata reflects the new name for the column but does not have an entry for the old name, so it would not be added to the temporary table.

For example:

```
var table = getTable("TableName").createNewTempTableBasedOnThis()
```

To create a temporary table with a custom definition, use the `createAsNewTempTable` method of `IBeforeUpgradeTable`. This method takes a `Pair` array in which the first object is a `String` defining the column name and the second object is an `IDataType` defining the column data type.

Renaming Tables

To rename a table use the `rename` function on the table object.

```
override function execute() {
    getTable("extTableName").rename("TableName_EXT")
}
```

Deleting Rows

To delete rows from a table, use the `delete` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `delete` method returns a delete builder (`IBeforeUpgradeDeleteBuilder`) that provides methods for comparing column data to restrict the rows that are deleted.

In the following example, all rows that have a columnA value of 0 are deleted.

```
var table = getTable("SomeTable")
var deleteBuilder = table.delete()
deleteBuilder.Query.compare("columnA", Equals, 0)
deleteBuilder.execute()
```

Inserting Rows

To insert rows of data use the `insert` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insert` method returns a builder (`IBeforeUpgradeInsertBuilder` for `IBeforeUpgradeTable`, `IInsertBuilder` for `IAfterUpgradeTable`) for SQL to perform an insert operation.

In the following example, an `IBeforeUpgradeInsertBuilder` is used to add two rows with three columns to table `myTable`. The `IBeforeUpgradeInsertBuilder` includes a description.

```
var myTable = getTable("SomeTable")
var insertBuilder = myTable.insert().withDescription("A custom insert
trigger to add two rows.")

insertBuilder
.mapColumn("columnA", "value of column A for first row")
.mapColumn("columnB", "value of column B for first row")
.mapColumn("columnC", "value of column C for first row")

insertBuilder.execute()

// add a second row
insertBuilder
.mapColumn("columnA", "value of column A for second row")
.mapColumn("columnB", "value of column B for second row")
.mapColumn("columnC", "value of column C for second row")

insertBuilder.execute()
```

Inserting Data Selected from Another Table

To insert data selected from another table use the `insertSelect` method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. The `insertSelect` method returns a builder (`IBeforeUpgradeInsertSelectBuilder` for `IBeforeUpgradeTable`, `IInsertSelectBuilder` for `IAfterUpgradeTable`). The builder includes a `mapColumn` method that can be passed explicit values, columns, or a query.

In the following example, the trigger sets `targetTable.column1` to an explicit value. The trigger sets `targetTable.column2` to the value of `sourceTable.sourceColumn`. Because there is no comparison being performed, the trigger will insert a row in the target table for each row in the source table:

```
var sourceTable = getTable("sourceTable")
var targetTable = getTable("targetTable")

var insertSelectBuilder = targetTable.insertSelect(sourceTable)

insertSelectBuilder.mapColumn("column1", "value") // sets a hard-coded value
.mapColumn("column2", sourceTable.getColumn("sourceColumn")) // sets column2 on target table to
// source table sourceColumn

insertSelectBuilder.execute()
```

In the next example, an existing table, `sourceTable`, is split into two tables, `targetTable1` and `targetTable2`.

```
var sourceTable = getTable("sourceTable")
var targetTable1 = getTable("targetTable1")
var targetTable2 = getTable("targetTable2")

var insertSelectBuilder1 = targetTable1.insertSelect(sourceTable)
var insertSelectBuilder2 = targetTable2.insertSelect(sourceTable)

insertSelectBuilder1.mapColumn("column1", sourceTable.getColumn("sourceColumn1"))
.mapColumn("column2", sourceTable.getColumn("sourceColumn2"))

insertSelectBuilder1.execute()
```

```
insertSelectBuilder2.mapColumn("column1", sourceTable.getColumn("sourceColumn3"))
    .mapColumn("column2", sourceTable.getColumn("sourceColumn4"))

insertSelectBuilder2.execute()
```

Updating Rows

To update rows in a table, use the update method of `IBeforeUpgradeTable` or `IAfterUpgradeTable`. This method returns a builder (`IBeforeUpgradeUpdateBuilder` or `IUpdateBuilder`). The builder includes methods to compare data to restrict which rows are updated.

In the following example, table `SomeTable` is updated to set `column1` to `SomeValue` for each row where the subtype matches a certain entity type:

```
var table = getTable("SomeTable")

// get IBeforeUpgradeUpdateBuilder
var ub = table.update()

// set column 1 to SomeValue
ub.set("column1", "SomeValue")
// where
    .compare("subType", Equals, getTypeKeyID(EntityType))
    .execute()
```

Running the Commission Payable Calculations Process

Run the Commission Payable Calculations process on your starting version before you upgrade the database. The Commission Payable Calculations process makes commission payable on direct bill policies.

To run the Commission Payable Calculations process

1. Start your pre-upgrade BillingCenter server.
2. Open BillingCenter and log in with an administrator account.
3. Open Server Tools by pressing ALT + SHIFT + T.
4. Click Batch Process Info.
5. In the Action column for Commission Payable Calculations, click Run. Wait for the process to complete before upgrading BillingCenter.

For more information about the Commission Payable Calculations process, see “Commission Payable Calculations Batch Processing” on page 126 in the *System Administration Guide*.

Configuring the Database Upgrade

You can set parameters for the database upgrade in the `BillingCenter 8.0.4 database-config.xml` file. The `<database>` block in `database-config.xml` contains parameters for database configuration, such as connection information. The `<database>` block contains an `<upgrade>` block that contains configuration information for the overall database upgrade. The `<upgrade>` block also contains a `<versiontriggers>` element for configuring general version trigger behavior and can contain `<versiontrigger>` elements to configure each version trigger.

This topic describes the parameters you can set for the database upgrade. For general database connection parameters, see “Deploying BillingCenter to the Application Server” on page 79 in the *Installation Guide*.

Adjusting Commit Size for Encryption

You can adjust the commit size for rows requiring encryption by setting the `encryptioncommitsize` attribute to an integer in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade encryptioncommitsize="10000">
    ...
  </upgrade>
</database>
```

If BillingCenter encryption is applied on one or more attributes, the BillingCenter database upgrade commits batches of encrypted values. The upgrade commits `encryptioncommitsize` rows at a time in each batch. The default value of `encryptioncommitsize` varies based on the database type. For Oracle, the default is 10000. For SQL Server, the default is 100.

Test the upgrade on a copy of your production database before attempting to upgrade the actual production database. If the encryption process is slow, and you cannot attribute the slowness to SQL statements in the database, try adjusting the `encryptioncommitsize` attribute. After you have optimized performance of the encryption process, use that `encryptioncommitsize` when you upgrade your production database.

Configuring Version Trigger Elements

The database upgrade executes a series of version triggers that make changes to the database to upgrade between versions. You can set some configuration options for version triggers in `database-config.xml`. Normally, the default settings are sufficient. Change these settings only while investigating a slow database upgrade.

The `<database>` element in `database-config.xml` contains an `<upgrade>` element to organize parameters related to database upgrades. Included in the `<upgrade>` element is a `<versiontriggers>` element, as shown below:

```
<database ...>
  <param ... />
  <upgrade>
    <versiontriggers dbmsperfinfotreshold="600" />
  </upgrade>
</database>
```

The `<versiontriggers>` element configures the instrumentation of version triggers. This element has one attribute: `dbmsperfinfotreshold`. The `dbmsperfinfotreshold` attribute specifies for all version triggers the threshold after which the database upgrader gathers performance information from the database. You specify `dbmsperfinfotreshold` in seconds, with a default of 600. If a version trigger takes longer than `dbmsperfinfotreshold` to execute, BillingCenter:

- queries the underlying database management system (DBMS).
- builds a set of html pages with performance information for the interval in which the version trigger was executing.
- includes those html pages in the upgrader instrumentation for the version trigger.

You can completely turn off the collection of database snapshot instrumentation for version triggers by setting the `dbmsperfinfotreshold` to 0 in `config.xml`.

The `<versiontriggers>` element can contain optional `<versiontrigger>` elements for each version trigger. Each `<versiontrigger>` element can contain the following attributes.

Attribute	Type	Description
<code>name</code>	String	The case-insensitive name of a version trigger.
<code>extendedquerytracingenabled</code>	Boolean	Oracle only. Controls whether or not to enable extended sql tracing (Oracle event 10046) for the SQL statements that are executed by the version trigger. Default is <code>false</code> . The output can be very useful when debugging certain types of performance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation.
<code>parallel-dml</code>	Boolean	Oracle only. See “Configuring Parallel DML and DDL Statement Execution” on page 272.
<code>queryoptimizertracingenabled</code>	Boolean	Oracle only. Controls whether or not to enable query optimizer tracing (Oracle event 10053) for the SQL statements that are executed by the version trigger. Default is <code>false</code> . The output can be very useful when debugging certain types of performance problems. Trace files that are generated only exist on the database machine. They are not integrated into the upgrade instrumentation.
<code>recordcounters</code>	Boolean	Controls whether the DBMS-specific counters are retrieved at the beginning and end of the use of the version trigger. Default is <code>false</code> . If true, then BillingCenter retrieves the current state of the counters from the underlying DBMS at the beginning of execution of the version trigger. If the execution of the version trigger exceeds the <code>dbmsperfinfotreshold</code> , then BillingCenter retrieves the current state of the counters at the end of the execution of the version trigger. BillingCenter writes differences to the DBMS-specific instrumentation pages of the upgrade instrumentation.
<code>updatejoinorderedhint</code>	Boolean	Oracle only. Whether to use the ORDERED hint for the UPDATE of a join. Default is <code>false</code> .
<code>updatejoinusemergehint</code>	Boolean	Oracle only. Whether to use the USE_MERGE hint for the UPDATE of a join. Default is <code>false</code> .
<code>updatejoinusenlhint</code>	Boolean	Oracle only. Whether to use the USE_NL hint for the UPDATE of a join. Default is <code>false</code> .

Deferring Creation of Nonessential Indexes

You can configure the upgrade to defer creation of nonessential indexes during the upgrade process until the upgrade completes and the application server is online. Nonessential indexes are performance-related indexes that do not enforce constraints. Creation of nonessential indexes can add significant time to the upgrade duration, so it is possible to defer this process. By default, the upgrade does not defer creation of these indexes.

To configure the upgrade to defer creation of nonessential indexes set the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` to `true`.

```
<database ...>
  <upgrade defer-create-nonessential-indexes="true">
    ...
  </upgrade>
</database>
```

If you opt to defer creation of nonessential indexes, BillingCenter runs the `DeferredUpgradeTasks` batch process as soon as the upgrade completes and the server is completely started. The `DeferredUpgradeTasks` batch process creates the nonessential performance indexes. The database user must have permission to create indexes until after the `DeferredUpgradeTasks` batch process is complete.

Deferring nonessential index creation can shorten the duration of the upgrade process. The BillingCenter database is then available sooner for tasks including upgrade verification and backing up the upgraded database before the database is opened up for production use. To take advantage of this earlier availability, perform upgrade testing and validation tasks while the `DeferredUpgradeTasks` batch process is running. Do not go into full production while the process is still running. The lack of so many performance-related indexes could likely make the system unusable.

Until the `DeferredUpgradeTasks` batch process has run to completion, BillingCenter reports errors during schema validation when starting. These include errors for column-based indexes existing in the data model but not in the physical database and mismatches between the data model and system tables.

Check the status of the `DeferredUpgradeTasks` batch process to determine when it has completed successfully. You can find the status of the deferred upgrade in the upgrade logs and on the BillingCenter [Upgrade Info](#) page. If the `DeferredUpgradeTasks` batch process fails, manually run the batch process again during non-peak hours.

If you do not opt to defer creation of nonessential indexes, BillingCenter creates these indexes as part of the upgrade process that must complete before the application server is online. If you do not want to defer creating nonessential indexes, the `defer-create-nonessential-indexes` attribute on the `<upgrade>` element in `database-config.xml` must be set to `false`. This is the default setting.

Configuring the Upgrade on Oracle

Configuring Column Removal

The database upgrade removes some columns. For Oracle, you can configure whether the removed columns are dropped immediately or are marked as unused. Marking a column as unused is a faster operation than dropping the column immediately. However, because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments. You can drop the unused columns after the upgrade during off-peak hours to free the space. Or, you can configure the database upgrade to drop the columns immediately during the upgrade. By default, the BillingCenter database upgrade marks columns as unused.

To configure the BillingCenter upgrade to drop columns immediately during the upgrade, set the `deferDropColumns` attribute of the `<upgrade>` block in `database-config.xml` to `false`. For example:

```
<database ...>
  ...
  <upgrade deferDropColumns="false">
    ...
  </upgrade>
</database>
```

By default, `deferDropColumns` is `true`.

Configuring Parallel DML and DDL Statement Execution

You can configure whether the upgrade executes DML (Data Manipulation Language) and DDL (Data Definition Language) statements in parallel or not and the degree of parallelism to use.

The `<upgrade>` element includes an `ora-parallel-dml` attribute. This attribute can be set to `disable`, `enable`, or `enable-all`. The default value is `enable`. If `ora-parallel-dml` is set to `disable`, the upgrade does not conduct parallel execution of DML statements. If `ora-parallel-dml` is set to `enable`, the upgrade executes DML statements in parallel if configured or coded for a version trigger. If `ora-parallel-dml` is set to `enable-all`, the upgrade executes DML statements in parallel in all cases unless turned off in the code or configuration for a version trigger.

The Boolean attribute `parallel-dml` of a `<versiontrigger>` element controls parallel execution for that version trigger. If `parallel-dml` is not set, the upgrade executes parallel DML statements if coded or if `ora-parallel-dml` is set to `enable_all` on the `<upgrade>` element. If `parallel-dml` is set to `false`, the upgrade does not execute DML statements in parallel. If `parallel-dml` is set to `true`, the upgrade executes DML statements in parallel if `ora-parallel` is set to `enable` or `enable_all`.

To configure the degree of parallelism for insert, update and delete operations, set the `degree-of-parallelism` attribute on the `<upgrade>` element. To configure the degree of parallelism for commands such as creating an index and enabling constraints using the alter table command, set the `degree-parallel-ddl` attribute on the `<upgrade>` element.

You can specify a value from 2 to 1000 to force that degree of parallelism. Specify a value of 1 to disable the use of parallel execution.

Setting either parameter to 0 configures BillingCenter to defer to Oracle to determine the degree of parallelism for the operations that attribute configures. The Oracle automatic parallel tuning feature determines the degree based on the number of CPUs and the value set for the Oracle parameter `PARALLEL_THREADS_PER_CPU`.

The default for both attributes is 4.

You can configure parallel DML execution on the `InsertSelectBuilder`, `BeforeUpgradeUpdateBuilder` and `BeforeUpgradeInsertSelectBuilder` of a custom version trigger using the `withParallelDml(boolean)` method. If not explicitly set to `true` or `false`, the upgrade uses parallel execution if configured. If set to `false`, the upgrade does not use parallel execution unless set to `true` for that version trigger. If set to `true`, it will be done unless set to `false` for that version trigger or `ora-parallel-dml` is set to `disable`.

Collecting Tablespace Usage and Object Size

To enable collection of tablespace usage and object size data on Oracle, set the `collectstorageinstrumentation` attribute of the `<upgrade>` block to `true`. For example:

```
<database ...>
  ...
  <upgrade collectstorageinstrumentation="true">
  ...
</upgrade>
</database>
```

A value of `true` enables BillingCenter to collect tablespace usage and size of segments such as tables, indexes and LOBs (large object binaries) before and after the upgrade. The values can then be compared to find the utilization change caused by the upgrade.

Disabling Oracle Logging

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="true">
  ...
</upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run statements with the `NOLOGGING` option.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. Some examples include Reporting, Disaster Recovery through Standby databases and Oracle Dataguard. To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

Disabling Statistics Update for the Database

Generating table statistics during upgrade is optional for Oracle databases. The overall time required to upgrade the database is shorter if the database upgrade does not update statistics. To disable statistics generation during the upgrade, set the `updatestatistics` attribute of the `<upgrade>` element to `false`:

```
<upgrade updatestatistics="false">
```

If `updatestatistics` is `true`, the upgrade updates statistics and deletes histograms on columns for which BillingCenter does not generate statistics. This setting enables the upgrade to update statistics on changed objects. It also configures BillingCenter to maintain column-level statistics consistent with what is allowed in the code, data model and configuration.

If statistics are not updated during the upgrade, BillingCenter reports a warning that recommends that you run the database statistics batch process in incremental mode. Additionally, the [Upgrade Info](#) page shows that statistics were not updated as part of the upgrade. If statistics generation was not disabled, the [Upgrade Info](#) page reports the runs of the statistics batch process, including incremental runs.

You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded BillingCenter in a production environment. If you defer generating statistics during the upgrade, Guidewire recommends that you generate full statistics as soon as possible after the upgrade. For instructions, see “[Commands for Updating Database Statistics](#)” on page 40 in the *System Administration Guide*.

The [Upgrade Info](#) page does not identify the following case: You ran an upgrade with `updatestatistics=true` after running a previous upgrade with `updatestatistics=false`, but you did not update statistics first.

When you click the **Download** button on the [Upgrade Info](#) page, you get a more detailed report. This report shows the value of the `updatestatistics` attribute at the time of upgrade. Additionally, the report shows the update statistics SQL statements that were skipped as part of the upgrade. These statements are provided for reference. You typically do not need to review these statements if you run the incremental database statistics process following the upgrade.

Disabling Statistics Update for Tables with Locked Statistics

If you have tables that have locked statistics, specify to keep statistics on these tables before starting the database upgrade. To specify to keep statistics on a table, set the `action` attribute of the `<tablestatistics>` element for that table to `keep`. The `<tablestatistics>` element is nested within the `<databasestatistics>` element, which is within the `<database>` element in `database-config.xml`.

For example, if statistics are locked on `bc_someTable_EXT`, specify a `<tablestatistics>` element for that table with the `action` attribute set to `keep`:

```
<database>
  ...
  <databasestatistics>
    <tablestatistics name="bc_someTable_EXT" action="keep" />
  </databasestatistics>
</database>
```

Configuring the Upgrade on SQL Server

Disabling SQL Server Logging

You can disable logging of direct insert and create index operations during the database upgrade by setting `allowUnloggedOperations` to `true` in the `<upgrade>` block. For example:

```
<database ...>
  ...
  <upgrade allowUnloggedOperations="true">
    ...
  </upgrade>
</database>
```

Setting `allowUnloggedOperations` to `true` causes the upgrade to run with minimal logging. This can improve the performance of the upgrade. During the upgrade, set the SQL Server recovery model to Simple or Bulk logged. Once the upgrade and deferred upgrade tasks are complete, you can revert the recovery model setting and back up the full database.

Although Guidewire recommends that you backup the database before and after the upgrade, there could be reasons to log all operations. If you require full logging due to the presence of solutions such as Database Mirroring, continue to use the Full recovery model and set `allowUnloggedOperations` to `false`.

To enable logging of direct insert and create index operations, set `allowUnloggedOperations` to `false`. If not specified, the default value of `allowUnloggedOperations` is `false`.

Storing Temporary Sort Results in tempdb

For SQL Server databases, you can specify to store temporary sort results in tempdb by setting the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` block to `true`. By using tempdb for sort runs, disk input and output is typically faster, and the created indexes tend to be more contiguous. By default, `sqlserverCreateIndexSortInTempDB` is `false` and sort runs are stored in the destination filegroup.

If you set `sqlserverCreateIndexSortInTempDB` to `true`, you must have enough disk space available to tempdb for the sort runs, which for the clustered index include the data pages. You must also have sufficient free space in the destination filegroup to store the final index structure, because the new index is created before the old index is deleted. Refer to <http://msdn.microsoft.com/en-us/library/ms188281.aspx> for details on the requirements to use tempdb for sort results.

Specifying Filegroup to Store Sort Results for Clustered Indexes

For SQL Server databases, a version trigger recreates non-clustered backing indexes for primary keys as clustered indexes.

Before recreating the indexes, the version trigger automatically drops (and later rebuilds) any referencing foreign keys and drops any clustered indexes on tables with a primary key.

If you are using filegroups, the upgrade recreates the clustered index in the OP filegroup. By default, the upgrade also stores the intermediate sort results that are used to build the index in the OP filegroup. You can configure the upgrade to instead use the tempdb filegroup for the intermediate sort results.

If you want the upgrade to stores the intermediate sort results in the tempdb filegroup, set the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` element to `true`.

```
<database ...>
  ...
  <upgrade sqlserverCreateIndexSortInTempDB="true" />
  ...
</upgrade>
</database>
```

This option increases the amount of temporary disk space that is used to create an index. However, it might reduce the time that is required to create or rebuild an index when tempdb is on a different set of disks from that of the user database.

By default, `sqlserverCreateIndexSortInTempDB` is `false`.

Downloading Database Upgrade Instrumentation Details

The database upgrade deletes upgrade instrumentation information for prior database upgrades. If the database upgrade detects any prior upgrade instrumentation data, it reports a warning and deletes the data. If you have run previous database upgrades, and you want to preserve upgrade instrumentation details, follow the procedure in “Viewing Detailed Database Upgrade Information” on page 314.

Checking the Database Before Upgrade

The upgrade runs a series of version checks prior to making any changes to the database. These version checks ensure that the database is in a state that can be upgraded. Guidewire includes a number of version checks with BillingCenter and you can also add custom version checks.

You can configure BillingCenter to run the version checks only, including custom version checks. Before upgrading the production database, run version checks on a clone of your production database to identify any issues with your data.

To run version checks without database upgrade

1. Start Studio for BillingCenter 8.0.4 by running the following command from the bin directory:
`gwbc studio`
2. Expand **configuration** → **config** and open **database-config.xml**.
3. Add the attribute `versionchecksonly=true` to the `database` element. The `versionchecksonly` attribute overrides the `autoupgrade` attribute. If both are set to true, BillingCenter only runs version checks when the server starts.
4. Verify that the database connection is pointing to a clone of your production database.
5. Save your changes.
6. Start the server.

BillingCenter reports the number of version check errors. For any errors reported BillingCenter reports which version check resulted in the error along with the error message.

If BillingCenter reports version check errors, fix the data and rerun the version checks. Repeat this process until no errors are reported on the production clone. Apply the fixes to your production database prior to upgrade.

With `versionchecksonly=true` set, BillingCenter runs all version checks regardless of a failure in one of the checks. During a regular upgrade, BillingCenter stops the upgrade if an error is detected.

After you have fixed all version check errors, set `versionchecksonly` to `false` to run the actual upgrade.

Disabling the Scheduler

Before you start the server to upgrade the database, disable the scheduler for batch processes and work queues. Disabling the scheduler prevents batch processes and work queues from launching immediately after the database upgrade.

To disable the scheduler

1. Open the BillingCenter 8.0.4 `config.xml` file in a text editor.
2. Set the `SchedulerEnabled` parameter to `false`.
`<param name="SchedulerEnabled" value="false"/>`
3. Save `config.xml`.

After you have successfully upgraded the database, you can enable the scheduler by setting `SchedulerEnabled` to `true`. This can be accomplished by performing the database upgrade using a WAR or EAR file that has the `SchedulerEnabled` parameter to `false`. After the upgrade is complete and verified, stop the server and deploy a new WAR or EAR file that differs from the first only by having `SchedulerEnabled` set to `true`. Finally, restart the server to activate the scheduler.

Suspending Message Destinations

Suspend all event message destinations before you upgrade the database to prevent BillingCenter from sending messages until you have verified a successful database upgrade.

To suspend message destinations

1. Start the BillingCenter server for the pre-upgrade version.
2. Log in to BillingCenter with an account that has administrative privileges, such as the superuser account.
3. Click the **Administration** tab.
4. Click **Event Messages**.
5. Select the check box to the left of the **Destination** column to select all message destinations.
6. Click **Suspend**.

Resume messaging after you have verified a successful database upgrade.

Starting the Server to Begin Automatic Database Upgrade

The database upgrade is an automatic process that occurs as you start the server with the upgraded configuration of a new BillingCenter version. The database upgrade normally completes in a few hours or less.

If the database upgrade stops before completing, then restore your database from the backup, correct any issues reported, and repeat the database upgrade.

IMPORTANT Before starting the upgrade, update database server software and operating systems as needed to meet the installation requirements of BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <https://guidewire.custhelp.com/app/resources/products/platform>.

WARNING Except for your first database upgrade trials, do not start the server until you have upgraded all rules. Otherwise, default validation rules execute. This could strand objects at a high validation level and make it impossible to edit parts of the object.

WARNING The database upgrade runs a series of version checks prior to making any changes. If any of these checks fail, the upgrade aborts and reports an error message. You can fix the issue, create an updated backup of the database and attempt the upgrade again without restoring from a backup. However, if you experience a failure during the version triggers or upgrade steps portion of the upgrade, refresh the database from a backup before attempting the upgrade again.

Before you start the server to begin the database upgrade, follow the procedure in “Updating Rounding Mode Parameter” on page 125.

Test the Database Upgrade

Prior to attempting the database upgrade on a full-production database clone, test the database upgrade.

To test the database upgrade

1. Connected to the built-in Quickstart database, successfully start the built-in Quickstart application server with a merged configuration data model, including merged extensions, data types, field validators, and so forth.
2. Connected to an empty database on an Oracle or SQL Server database server, successfully start the Quickstart application server from the preceding step.
3. Connected to a restored backup of a production clone, start either the same Quickstart server from the preceding step or a supported third-party application server with your custom configuration.

In a development environment the database upgrade process records checkpoints of upgrade triggers that complete successfully. You can restart a failed database upgrade, and it resumes with the upgrade trigger that failed. This restart feature helps you test the upgrade with realistically large data sets. You avoid time spent to restore the database and rerun upgrade triggers that worked successfully.

Guidewire provides this feature for convenience while testing. However, it does not work for all failure scenarios. Even in development mode, under certain scenarios, you will have to restore a backup of the database taken prior to the upgrade attempts and then run the upgrade.

The database upgrade writes SQL executed by the failed trigger to the console. To restart a test database upgrade from a checkpoint reached in an earlier upgrade, manually roll back any database changes that occurred during the upgrade trigger that failed. Resolve the problem that caused the trigger to fail. Then start the server again to restart the upgrade. The upgrade skips successful upgrade triggers and continues by rerunning the trigger that failed.

A test run of your upgrade is successful only when it runs from start to finish without a restart.

WARNING Never use the restart feature of database upgrade in a production environment.

Integrations and Starting the Server

Disable all integrations during the automatic database upgrade. Integration points might require updates due to changes in Guidewire APIs. See the *BillingCenter New and Changed Guide* for specifics.

It is not necessary to have completely migrated integrations before attempting to start the server for the first time. If you have integrations that rely on non-Guidewire applications, do not expect these integrations to work the first time you start the server.

Understanding the Automatic Database Upgrade

As the database upgrade proceeds, it logs messages to the console as well as the log file describing its progress. The database upgrade process requires thousands of steps, divided into three phases. Due to the relational nature of a database, these phases must execute in a specific order for the upgrade to succeed.

During the first phase, the upgrader first executes custom `BeforeUpgradeVersionTrigger` version checks and triggers defined in the `IDataModelUpgrade` plugin. The upgrader next runs version checks defined by Guidewire. Then, the upgrader uses a set of version triggers defined by Guidewire to determine the actions that are required. The database upgrader requires version triggers in order to perform the following types of tasks:

- changing a datatype (other than just length)
- migrating data
- dropping a column
- dropping a table
- renaming a column
- renaming a table

Specific version triggers are described in this topic.

Many version triggers have version checks associated with them. These checks ensure that the database is ready for the associated version trigger. The database upgrade runs all checks before running any version triggers. If a check detects a problem, it reports the issue, including a sample SQL query to find specific problematic records. If a version check discovers an issue, the database upgrade stops before any version triggers are run. Therefore, it is not necessary to restore the database from a backup if a version check reports an error. Correct the issue and then create a new backup of the database. Then, if you encounter errors after the version check stage, you can restore a version of your database with the issue reported by the version check resolved.

In the second phase, the upgrader compares the target data model and the current database to determine how they differ. The upgrader makes changes to the database that do not require a version trigger during this phase.

Following this process, the third phase runs a subsequent set of version triggers. These triggers create actions that must be run last due to a dependency on an earlier phase.

After the database upgrade concludes, it reports issues that the upgrader encountered and did not complete.

You are responsible for correcting these issues. This might involve modifying the data model or altering the table manually. If you do not correct them, the next time you start the server you do *not* see a message that the database and the data model are out of sync. You must then use the `system_tools` command to verify the database schema.

Note: Given the complexity of database upgrade, Guidewire does not expose specific upgrade actions/steps to clients either in SQL or Java form. Any manual attempts to recreate or control the upgrade process can result in problems in the BillingCenter database. Recovery from such attempts is not supported.

Version Trigger Descriptions

The database upgrade uses version triggers to perform the actions described by sections within this topic. If a version trigger has an associated version check, the check is described with the trigger. Review these descriptions to familiarize yourself with some of the changes and to understand version checks. If a version check reports an issue, review the error message and consult the description of the relevant version trigger for more information.

Converting Primary Key Indexes to Clustered Indexes

For SQL Server databases, this step recreates non-clustered backing indexes for primary keys as clustered indexes.

Before recreating the indexes, the upgrade automatically drops (and later rebuilds) any referencing foreign keys and drops any clustered indexes on tables with a primary key.

If you are using filegroups, the upgrade recreates the clustered index in the OP filegroup. By default, the upgrade also stores the intermediate sort results that are used to build the index in the OP filegroup. You can configure the upgrade to instead use the tempdb filegroup for the intermediate sort results.

If you want the upgrade to store the intermediate sort results in the tempdb filegroup, set the `sqlserverCreateIndexSortInTempDB` attribute of the `upgrade` element to `true`.

```
<database ...>
  ...
  <upgrade sqlserverCreateIndexSortInTempDB="true" />
  ...
</upgrade>
</database>
```

This option increases the amount of temporary disk space that is used to create an index. However, it might reduce the time that is required to create or rebuild an index when tempdb is on a different set of disks from that of the user database.

By default, `sqlserverCreateIndexSortInTempDB` is `false`.

Updating TableRegistryEntry and EncryptedColumnRegistryEntry to Use Lowercase

The upgrade sets all `TableName` values in the `bc_TableRegistry` and `bc_EncryptedColumnRegistry` tables to lowercase values.

Dropping Upgrade Instrumentation Tables

The database upgrade drops the following upgrade instrumentation tables:

- `bc_purgeerror`
- `bc_purgehistory`
- `bc_purgerecord`
- `bc_upgradecuststatement`
- `bc_upgradeencryptchunk`
- `bc_upgradeencryptstep`
- `bc_upgradephase`
- `bc_upgradephasedbmsdump`
- `bc_upgradestep`
- `bc_upgradeversiontrigger`
- `bc_upgradevtstatement`

The database upgrade also drops the `bc_upgradevtdbmsdump.UpgradeVersionTriggerID` column.

Migrating LoadErrorRow.RowNumber to Larger Data Type

On SQL Server databases, the upgrade converts `LoadErrorRow.RowNumber` to a `bigint` datatype.

Converting ID Columns to 64-bit Numbers

The upgrade converts the following columns to 64-bit numbers to make a much greater number of possible IDs available to BillingCenter for their parent entities:

- `bc_loaderrorrow_RowNumber`

Adding New Index to TroubleTicketJoinEntity

The upgrade adds a new index to `bc_troubleticketjoinentity`. The index is defined in `config/metadata/TroubleTicketJoinEntity.eti`.

Converting and Removing BCContact Subtype

In versions of BillingCenter prior to 7.0, BillingCenter had a subtype of the Guidewire platform `Contact` entity named `BCContact`. `BCContact` had a typelist field named `Type` which could have one of two typekey values: `ContactType.TC_COMPANY` or `ContactType.TC_PERSON`.

BillingCenter 7.0 and newer uses `Contact` instead of `BCContact` and uses the `Company` and `Person` subtypes of `Contact`.

The upgrade does the following:

- Changes all rows in the `bc_Contact` table that had a Subtype of `BCContact` and a Type of `ContactType.TC_COMPANY` to instead have a Subtype of `Company`.
- Sets to null the `FirstName` and `LastName` of any `Contact` that is now subtype `Company`.
- Changes all rows in the `bc_Contact` table that had a Subtype of `BCContact` and a Type of `ContactType.TC_PERSON` to instead have a Subtype of `Person`.
- Drops the `bct1_ContactType` typelist table.
- Drops the `bc_Contact.Type` column.

Deleting Legacy Payment Work Item Table

BillingCenter 2.1 included a Legacy Payment batch process. This batch process was removed in BillingCenter 3.0. The upgrade drops the `bc_LegacyPaymentWorkItem` table as it is no longer in use.

Refactoring Write-offs

The upgrade refactors write-offs. The upgrade performs the following actions:

- Renames `bc_ProducerContext.DistItem` to `bc_ProducerContext.DistItemID`.
- Renames `bc_ZeroCmsnEarnedMarker.BasePaymentItem` to `bc_ZeroCmsnEarnedMarker.BasePaymentItemID`.
- Changes the `ChargeWriteoff` subtype to `ChargeGrossWriteOff`.
- Upgrades `ChargeCommissionWriteoff` entities to a subtype of `WriteOff`. The upgrade migrates the data from `bc_ChargeCmsnWtoff` to `bc_Writeoff`.
- Sets `bc_CmsnReduction.CommissionWriteoffID` to point to the new `Writeoff`.
- Drops the `ChargeCommissionWriteoffID` column from `bc_CmsnReduction`.
- Sets `bc_AuthorityEvent.WriteoffID` to point to the new `Writeoff` for authority events that currently have a Subtype of `ChargeCommissionWriteOffAuthorityEvent`.
- Sets `bc_AuthorityEvent.Subtype` to `WriteOffAuthEvent` for authority events that currently have a Subtype of `ChargeCommissionWriteOffAuthorityEvent`.
- Drops the `ChargeCmsnWriteoffID` column from `bc_AuthorityEvent`.
- Sets `bc_Activity.WriteoffID` to point to the new `Writeoff` for activities that currently have a Subtype of `ChargeCommissionWriteOffApprovalActivity`.
- Sets `bc_Activity.Subtype` to `WriteoffApprActivity` for activities that currently have a Subtype of `ChargeCommissionWriteOffApprovalActivity`.
- Drops the `ChargeCommissionWriteoffID` column from `bc_Activity`.
- Drops the `bc_ChargeCmsnWtoff` table.

Refactoring Modifying

Removing Modifying Money

The database upgrade removes all “modifying” subtypes of `BaseMoneyReceived`, and rekeys them to the original subtype. The upgrade performs the following actions:

- Renames the `bc_OriginalPMR` edge table to `bc_MovedFromPMR`.
- Drops the `bc_ModifiedPMR` table.
- Creates the `bc_ModifiedFromPMR` edge table and populates it with rows from the following edge tables:
 - `bc_moddbpmtmoney`
 - `bc_modpymtmon`
 - `bc_modzdamr`
 - `bc_modzddmr`
- Rekeys `BaseMoneyReceived` subtypes:

Old subtype	New subtype
ModifyingDirectMoney	DirectBillMoneyRcvd
ModifyingAgencyMoney	AgencyBillMoneyRcvd
ModifyingZeroDollarDMR	ZeroDollarDMR
ModifyingZeroDollarAMR	ZeroDollarAMR

Converting Modifying DistItems

The database upgrade converts “modifying” `DistItem` entities into `DistItem` entities. The upgrade performs the following actions:

- Adds a new edge foreign key table, `bc_ModifiedFromDI`.
- Copies values into `bc_ModifiedFromDI`.
- Removes the following old edge foreign key tables:
 - `bc_ModDbPmntItem`
 - `bc_ModPmntItem`
 - `bc_ModPromItem`
- Rekeys `bc_BaseDistItem` subtypes into their non-modifying types.

Old subtype	New subtype
<code>ModifyingAgencyPmntItem</code>	<code>AgencyPaymentItem</code>
<code>ModifyingDirectPmntItem</code>	<code>DirectBillPaymentItem</code>
<code>ModifyingPromiseItem</code>	<code>AgencyPromiseItem</code>

Converting Modifying SuspenseItems

The database upgrade converts “modifying” `SuspenseItem` entities into regular `SuspenseItem` entities. The upgrade performs the following actions:

- Adds a new edge foreign key table, `bc_ModifiedFromSPI`.
- Copies values into `bc_ModifiedFromSPI`.
- Removes the following old edge foreign key tables:
 - `bc_ModSuspPmntItem`
 - `bc_ModDBSuspPmntItem`
- Rekeys `bc_BaseNonRecDistItem` subtypes into their non-modifying types.

Old subtype	New subtype
<code>ModifyingDirectSuspPmntItem</code>	<code>DirectSuspPmntItem</code>
<code>ModifyingAgencySuspPmntItem</code>	<code>AgencySuspPmntItem</code>

Removing Modifying Subtypes of BaseDists

The database upgrade removes all “modifying” subtypes of `BaseDist`, and rekeys them to the original subtype. The upgrade performs the following actions:

- Creates a `bc_ModDist` edge table.
- Populates the new `bc_Moddist` edge table with rows from the following edge tables:
 - `bc_ModAgcyPmnt`
 - `bc_ModAgcyProm`
 - `bc_ModDBPmnt`
- Rekeys `BaseMoneyReceived` subtypes:

Old subtype	New subtype
<code>ModifyingAgencyPayment</code>	<code>AgencyCyclePayment</code>
<code>ModifyingAgencyPromise</code>	<code>AgencyCyclePromise</code>
<code>ModifyingDirectPayment</code>	<code>DirectBillPayment</code>

Converting HiddenOverridingDefaultPayer to OverridingPayerContainer

This step performs the following actions:

- Renames bc_Charge column HiddenTAccountContainerID to OverridingPayerContainerID.
- Adds column OverridingPrimaryCmsnRcvrID to bc_Charge and populates it with the ProducerCode when OverridingPayer is a ProducerCode.
- Changes the value of bc_Charge.OverridingPayerContainerID to reference a Producer instead of a ProducerCode.

Refactoring Reversed Dist Items

The database upgrade refactors reversed dist items.

Splitting the BaseDist DistItems Array

This step splits the BaseDist.DistItems array into DistItems and ReversedDistItems arrays, by performing the following actions:

- Adds ReversedDistID to BaseDistItem.
- Renaming BaseDistItem.BaseDistID to ActiveDistID.
- If BaseDistItem.ReversedDate is not null, set ReversedDistID equal to ActiveDistID.
- If BaseDistItem.ReversedDate is not null, set ActiveDistID equal to null.

Splitting the BaseDist NonReceivableDistItems Array

This step splits the BaseDist.NonReceivableDistItems array into NonReceivableDistItems and ReversedNonReceivableDistItems arrays, by performing the following actions:

- Adds ReversedDistID to NonReceivableBaseDistItem.
- Renames NonReceivableBaseDistItem.BaseDistID to ActiveDistID.
- If NonReceivableBaseDistItem.ReversedDate is not null, set ReversedDistID equal to ActiveDistID.
- If NonReceivableBaseDistItem.ReversedDate is not null, set ActiveDistID equal to null.

Changing the BaseDist to BaseMoneyReceived Relationship

In BillingCenter 7.0, the BaseDist to MoneyReceived relationship was refactored so that BaseMoneyReceived entities now point to BaseDist entities. BillingCenter 7.0 added a bc_ActiveMoneyRcvd edge foreign key table so that a BaseDist can point to its active MoneyReceived entity.

Adding Account Funds Tracking Tables

The database upgrade adds the following tables used in funds tracking, along with indexes for each table:

- bc_FundsAllotment
- bc_FundsTracker
- bc_FundsTrackingEnableMarker
- bc_PaymentItemAction
- bc_PaymentItemGroup

See “Account Funds Tracking” on page 241 in the *Application Guide* for information about the account funds tracking feature.

Dropping Invoice PaidStatus Column

The upgrade drops the bc_Invoice.PaidStatus column. You can use the following code to return the paid status of an invoice:

```
gw.plugin.invoice.impl.Invoice().getPaidStatus(invoice)
```

Converting Account Due Date Billing to Typekey

The database upgrade converts the bit column `bc_Account.DueDateBilling` to the typekey column `bc_Account.BillDateOrDueDateBilling`. The `BillDateOrDueDateBilling` column stores a typekey for the `BillDateOrDueDateBilling` typelist. These typekeys indicate whether invoice dates are computed from a fixed bill date or from a fixed due date. The `BillDateBilling` typekey indicates that the bill date is specified and the due date is computed from the specified bill date. The `DueDateBilling` typekey indicates that the due date is specified and the bill date is computed from the specified due date.

Adding Policy-level Billing

The upgrade adds columns used for policy-level billing. The upgrade adds a `bc_Account.BillingLevel` typekey column and sets it to `AccountLevelBilling`. The upgrade also adds a `bc_InvoiceStream.PolicyID` column.

See “Policy-Level Billing” on page 301 in the *Application Guide*.

Adding Custom Billing

The upgrade adds custom billing to BillingCenter. This includes adding columns to the `bc_InvoiceStream` table to allow invoice streams to override account default values for invoicing attributes.

The database upgrade first runs a version check that ensures:

- No `InvoiceStream` has a negative `DaysBeforeAnchorForBilling` value.
- No monthly `InvoiceStream` has a non-null `SecondAnchorDate`. Monthly invoice streams are expected to have only one anchor date.
- No weekly `InvoiceStream` has a non-null `SecondAnchorDate`. Weekly invoice streams are expected to have only one anchor date.
- No every-other-week `InvoiceStream` has a non-null `SecondAnchorDate`. Every-other-week invoice streams are expected to have only one anchor date.
- No `InvoiceStream` for an account that uses the responsive payment method overrides the payer’s lead time. This check ensures that there are no invoice streams for an account that uses the responsive payment method with `DaysBeforeAnchorForBilling` that is not 0 and not equal to `InvoiceStream.Account.BillingPlan.PaymentDueInterval`.
- No `InvoiceStream` for an account that uses the non-responsive payment method overrides the payer’s lead time. This check ensures that there are no invoice streams for an account that uses a non-responsive payment method with `DaysBeforeAnchorForBilling` that is not 0 and not equal to `InvoiceStream.Account.BillingPlan.NonResponsivePmntDueInterval`.
- No `InvoiceStream` for a producer overrides the payer’s lead time. This check ensures that there are no invoice streams for an account that uses the responsive payment method with `DaysBeforeAnchorForBilling` that is not 0 and not equal to `InvoiceStream.Producer.AgencyBillingPlan.PaymentTermInDays`.

If these checks all pass, the upgrade performs the following actions:

- Alters the `bc_InvoiceStream.Name` column to be nullable. This column was deprecated in BillingCenter 7.0.
- Rebuilds all indexes on `bc_InvoiceStream`.
- Adds `OverridingFirstAnchorDate` and `OverridingSecondAnchorDate` columns to `bc_InvoiceStream`. In versions prior to 7.0, invoicing attributes for an `InvoiceStream` were stored in the `bc_InvoiceStream` columns `firstAnchorDate`, `secondAnchorDate`, and `daysBeforeAnchorDateForInvoiceBilling`. In BillingCenter 7.0 and newer, `bc_InvoiceStream` has the nullable columns `overridingFirstAnchorDate` and `overridingSecondAnchorDate` instead. The dates are derived from the payer if the override is `null`. The `daysBeforeAnchorDateForInvoiceBilling` column is replaced by the nullable columns `overridingBillOrDueDateBilling` and `overridingInvoicingLeadTime`.
- For monthly streams, sets `OverridingFirstAnchorDate` to `FirstAnchorDate` if the account’s `InvoiceDayOfMonth` is different from the `FirstAnchorDate` day-of-month.

- For weekly streams, sets `OverridingFirstAnchorDate` to `FirstAnchorDate` if the account's `InvoiceDayOfWeek` is not the same weekday as the `FirstAnchorDate`.
- For every-other-week streams, sets `OverridingFirstAnchorDate` to `FirstAnchorDate` if the account's `EveryOtherWeekInvoiceAnchor` is different from `FirstAnchorDate`.
- For twice-per-month streams, sets `OverridingFirstAnchorDate` to `FirstAnchorDate` and `OverridingSecondAnchorDate` to `SecondAnchorDate` if the account's `FirstTwicePerMonthInvoiceDayOfMonth` is different from `FirstAnchorDate` day-of-month, or the account's `SecondTwicePerMonthInvoiceDayOfMonth` is different from `SecondAnchorDate` day-of-month.
- For producer monthly streams, sets the `OverridingFirstAnchorDate` to `FirstAnchorDate` if the producer's standard cycle close day-of-month is different from the `FirstAnchorDate` day-of-month.
- For producer non-monthly streams, sets the `OverridingFirstAnchorDate` to `FirstAnchorDate` and `OverridingSecondAnchorDate` to `SecondAnchorDate` for all producer streams with any periodicity other than monthly. By default, all producer streams are monthly, so any non-monthly producer streams must be using overriding anchor dates.
- For all account streams with custom periodicity, sets the `OverridingFirstAnchorDate` to `FirstAnchorDate` and `OverridingSecondAnchorDate` to `SecondAnchorDate`.
- Adds the `OverridingBillOrDueDateBilling` column to `bc_InvoiceStream`.
- Sets `OverridingBillOrDueDateBilling` to `DueDateBilling` if the account is using bill date billing and the stream is using due date billing. A BillingCenter 3.0 stream with `DaysBeforeAnchorForBilling` greater than 0 was using due date billing.
- Sets `OverridingBillOrDueDateBilling` to `BillDateBilling` if the account is using due date billing and the stream was using bill date billing. A BillingCenter 3.0 stream with `DaysBeforeAnchorForBilling` equal to 0 was using bill date billing.
- Sets `OverridingBillOrDueDateBilling` to `DueDateBilling` if the stream is a producer stream and the stream was using due date billing. All Producers use bill date billing. A BillingCenter 3.0 stream with `DaysBeforeAnchorForBilling` greater than 0 was using due date billing.
- Adds the `OverridingInvoicingLeadTime` column to `bc_InvoiceStream` and sets it to `null`.
- Adds the `OverridingPaymentInstrumentID` column to `bc_InvoiceStream` and sets it to `null`.
- Drops the `bc_InvoiceStream.DaysBeforeAnchorForBilling` column.
- Drops the `bc_InvoiceStream.FirstAnchorDate` column.
- Drops the `bc_InvoiceStream.SecondAnchorDate` column.
- Adds the `OverridingInvoiceStreamID` column to `bc_PolicyPeriod`.
- Adds the `OverridingInvoiceStreamID` column to `bc_Charge`.
- Adds the `CreationOrder` column to `bc_InvoiceStream`.

[Adding AccountPaymentPlan Table for List Bill Accounts](#)

This step adds a `bc_AccountPaymentPlan` table and indexes. The `bc_AccountPaymentPlan` table stores the relationship between an Account and a PaymentPlan. This table is used for list bill accounts.

For more information see “List Bill Accounts” on page 331 in the *Application Guide*.

[Fixing Spelling Issue for Payment Received](#)

BillingCenter versions prior to 7.0 could include a spelling error for the name, description and typecode of the `PaymentReceived` typekey in `bct1_WriteoffReversalReason`. The database upgrade corrects the spelling and updates references to the typekey ID in `bc_WriteOffReversal` if necessary.

Retiring Modified Distributions

The database upgrade retires all distribution records in `bc_BaseDist` that have an ID that corresponds to a `ForeignEntityID` on `bc_ModDist`.

Converting to Payment Instruments

BillingCenter 7.0 added payment instruments. A payment instrument is a financial instrument that can be or has been used to make a payment. If the payment instrument points at an account or a producer, the payment instrument can be used to request more payments.

The database upgrade removes payment method details columns from several entities. To preserve the information from these columns, see “Preserving Payment Method Details” on page 233.

Creating Singleton Payment Instruments

BillingCenter 7.0 and newer implement certain types of `PaymentInstrument` entities as system-wide singletons because they contain no uniquely identifying information. These types include `AccountUnapplied`, `Cash`, `Check`, `Responsive`, and `ProducerUnapplied`.

The database upgrade creates the `bc_PaymentInstrument` table and populates it with these singletons. The upgrade sets `bc_PaymentInstrument.PaymentMethod` to the typekey for the payment method and sets `bc_PaymentInstrument.Immutable` to true.

Setting Default Payment Instrument on Accounts

As of BillingCenter 7.0, accounts are required to have a `PaymentInstrument` with the `PaymentMethod Responsive`. Responsive means that the insured will be sent an invoice that requires a response in the form of a payment. The upgrade sets each account's `DefaultPaymentInstrument` field to point to the singleton `Responsive PaymentInstrument`.

Generating Payment Instruments for Accounts

The upgrade generates a `PaymentInstrument` based on the `PaymentMethod` set on each `Account`. The upgrade generates a `PaymentInstrument` for the `Account` if the `PaymentMethod` is `ach`, `creditcard`, or `misc`. The upgrade then drops the following payment method details columns from `bc_Account`:

- `BankABANumber`
- `BankAccountNumber`
- `BankAccountType`
- `BankName`
- `CreditCardExpDate`
- `CreditCardIssuer`
- `CreditCardNumber`
- `PaymentMethod`

Setting Default Payment Instrument on Producers

As of BillingCenter 7.0, producers are required to have a `PaymentInstrument` with the `PaymentMethod Responsive`. Responsive means that the insured will be sent an invoice that requires a response in the form of a payment. The upgrade sets each producer's `DefaultPaymentInstrument` field to point to the singleton `Responsive PaymentInstrument`.

Generating Payment Instruments for Producers

The upgrade generates a `PaymentInstrument` based on the `PaymentMethod` set on each `Producer`. The upgrade does not generate a `PaymentInstrument` if the `PaymentMethod` is `Responsive`, `Cash`, `Check`, or `ProducerUnapplied` because those are singleton `PaymentInstrument` types. The upgrade then drops the following payment method details columns from `bc_Producer`:

- BankABANumber
- BankAccountNumber
- BankAccountType
- BankName
- CreditCardExpDate
- CreditCardIssuer
- CreditCardNumber
- PaymentMethod

Converting Other Entities from Using `PaymentMethodDetails` to `PaymentInstrument`

The upgrade converts the following entities from using the old `PaymentMethodDetails` delegate columns to using `PaymentInstrument`:

- Disbursement
- IncomingProducerPayment
- OutgoingPayment
- PaymentMoneyReceived
- PaymentRequest
- SuspensePayment

As of BillingCenter 7.0, each of these entities must link to a `PaymentInstrument`. The upgrade generates a `PaymentInstrument` for each entity based on the `PaymentMethod` field, unless the `PaymentMethod` is for a singleton `PaymentInstrument`. If the `PaymentMethod` is for a singleton `PaymentInstrument`, the upgrade links the entity to the singleton `PaymentInstrument`. The upgrade then drops the following `PaymentMethodDetails` delegate columns from the entity:

- BankABANumber
- BankAccountNumber
- BankAccountType
- BankName
- CreditCardExpDate
- CreditCardIssuer
- CreditCardNumber
- PaymentMethod

Updating Statistics on `PaymentInstrument`

After generating payment instruments, the upgrade updates database statistics on the `bc_PaymentInstrument` table.

Dropping CreditCardIssuer Typelist

This step drops the `CreditCardIssuer` column from `bc_TmpPayment`. Previous upgrade steps removed `CreditCardIssuer` from other entities because the `CreditCardIssuer` column is part of the `PaymentMethodDetails` delegate.

Dropping PolicyPeriod PaymentMethod Column

This step drops the `bc_PolicyPeriod.PaymentMethod` column.

Adding PolicyPeriod OverridingPayerAccount Column

This step creates the `bc_PolicyPeriod.OverridingPayerAccountID` column with a `null` default value. This column links to the `Account` to use as the payer for new charges and items on the `PolicyPeriod`.

Converting PolicyPeriod AgencyBill Bit to BillingMethod TypeKey Column

This step converts the `bc_PolicyPeriod.AgencyBill` bit to a `bc_PolicyPeriod.BillingMethod` typekey column. The `BillingMethod` column is a typekey to the `PolicyPeriodBillingMethod` typelist. The upgrade sets `BillingMethod` to `AgencyBill` if `bc_PolicyPeriod.AgencyBill` was true. If `bc_PolicyPeriod.AgencyBill` was false, the upgrade sets `BillingMethod` to `DirectBill`.

Adding PolicyPeriod ConfirmationNotificationState Column

This step creates the `bc_PolicyPeriod.ConfirmationNotificationState` column. The `ConfirmationNotificationState` column is a typekey to the `ConfirmationNotification` typelist. The upgrade sets `ConfirmationNotificationState` to `DoNotNotify`.

Renaming BaseDist NegativeWriteoffAmount to WriteoffAmount

The upgrade renames the `bc_BaseDist` column `NegativeWriteoffAmount` to `WriteOffAmount` and changes the sign of the value stored in the column.

Upgrading Execution Date of WriteOff and NegativeWriteOff

The database upgrade performs the following actions:

- Renames `bc_WriteOff.WriteoffDate` to `ExecutionDate`.
- Sets `bc_Writeoff.ExecutionDate` for each non-reversal `ProdWriteoffContainer` to the `TransactionDate` of its earliest `Transaction`, if any.
- Sets `bc_Writeoff.ExecutionDate` for each reversal `ProdWriteoffContainer` to the `TransactionDate` of the `Writeoff` that it reversed.
- Creates a new column: `bc_NegativeWriteoff.ExecutionDate`.
- Sets `bc_NegativeWriteoff.ExecutionDate` for each non-reversal `NegativeWriteoff` to the `TransactionDate` of its earliest `Transaction`, if any.
- Sets `bc_NegativeWriteoff.ExecutionDate` for each reversal `NegativeWriteoff` to the `TransactionDate` of the `Writeoff` that it reversed.

Adding ChargeWrittenOffID to ProducerContext and ZeroCmsnEarnedMarker

This step adds a `ChargeWrittenOffID` column to `bc_ProducerContext` and `bc_ZeroCmsnEarnedMarker`.

Adding CommissionWriteOffID to CmsnReduction

This step adds a `CommissionWriteOffID` column to `bc_CmsnReduction`.

Upgrading Agency Bill Exception Fields

The database upgrade performs the following actions:

- Renames `bc_InvoiceItem` column `CarriedForwardDate` to `PaymentCarriedForwardDate`.
- Adds column `PromiseCarriedForwardDate` to `bc_InvoiceItem`.
- Sets the `PromiseCarriedForwardDate` on `bc_InvoiceItem` to the most recent `ExecutedDate` of the `BaseDistItem` for the `InvoiceItem` if the `AppliedDate` and `ReversedDate` of the `BaseDistItem` are null and `InvoiceItem.PromiseEligibleForException` is false.
- Sets `bc_InvoiceItem.PaymentCarriedForwardDate` to null for `InvoiceItem` records where `bc_InvoiceItem.PaymentEligibleForException` equals true.
- Sets `bc_InvoiceItem.HasBeenPaymentException` to true for `InvoiceItem` records where `bc_InvoiceItem.PaymentExceptionDate` is not null.

- Sets `bc_InvoiceItem.PaymentExceptionDate` to null for `InvoiceItem` records where `Amount` equals `PaidAmount` plus `GrossAmountWrittenOff` and `PrimaryCommissionAmount` equals `PrimaryPaidCommission` plus `PrimaryCmsnPayableAmount` plus `PrimaryWrittenOffCommission`.
- Sets `bc_InvoiceItem.PromiseExceptionDate` to null for `InvoiceItem` records where `Amount` equals `PromisedAndPaidAmount` plus `GrossAmountWrittenOff` and `PrimaryCommissionAmount` equals `PromisedCommission` plus `PrimaryPaidCommission` plus `PrimaryCmsnPayableAmount` plus `PrimaryWrittenOffCommission`.
- Drops the following columns from `bc_SnapshotInvoiceItem`:
 - `CarriedForwardDate`
 - `ExceptionComments`
 - `HasBeenPaymentException`
 - `PaymentEligibleForException`
 - `PaymentExceptionDate`
 - `PromiseEligibleForException`
 - `PromiseExceptionDate`
- Drops the following columns from `bc_InvoiceItem`:
 - `PaymentEligibleForException`
 - `PromiseEligibleForException`

Adding and Populating CreationOrder Column to InvoiceStream

The database upgrade adds a `CreationOrder` column to `bc_InvoiceStream`.

For a set of `InvoiceStream` entities with matching `AccountID`, `ProducerID`, `PolicyID`, `Periodicity`, and `Retired` columns, the upgrade sets a unique `CreationOrder`. The upgrade sets the lowest `CreationOrder` number on the `InvoiceStream` with the earliest `CreateTime`.

Adding Columns to Message and MessageHistory

The database upgrade adds the following foreign key columns to `bc_Message` and `bc_MessageHistory`:

- `AccountID`
- `ContactID`
- `PolicyPeriodID`
- `ProducerID`

Updating Producer Statement Sequence Numbers

The upgrade updates `bc_sequence` to set the producer statement sequence to the value of the transaction sequence if the producer statement number sequence is less than the transaction number sequence.

Replacing PolicyLevelPaymentOption Typecode

The upgrade replaces the `PolicyLevelPaymentOption` typecode `keepextrawithpolicy` with `distributeextra`. The upgrade then updates `bc_Account.PolicyLevelPaymentOption` to change instances with typecode for `keepextrawithpolicy` with the typecode for `distributeextra`.

This step applies to upgrades from versions prior to 3.0.1.

Resetting Work Item Tables

To accommodate data model changes to certain work item tables, the upgrade deletes all references to rows in the work item table. These references include process history, instrumented worker, and instrumented worker task. The upgrade then drops the table. The tables are later regenerated with the new data model.

The following tables are reset for upgrades from versions prior to 3.0.2.

Table	Associated process
bc_ActivityEscalWorkItem	ActivityEsc
bc_StatementBilledWorkItem	StatementBilled
bc_StatementDueWorkItem	StatementDue

Removing ExcessFund

The upgrade deletes the `bc_ExcessFundWorkItem` table and drops the `ExcessFunds` column from `bc_Plan`.

This step applies to upgrades from versions prior to 3.0.2.

Updating Account Due Date Billing

This step adds a `BillDateDueDateBilling` column to `bc_Account`. The `BillDateOrDueDateBilling` column stores a typekey for the `BillDateOrDueDateBilling` typelist. These typekeys indicate whether invoice dates are computed from a fixed bill date or from a fixed due date. The `BillDateBilling` typekey indicates that the bill date is specified and the due date is computed from the specified bill date. The `DueDateBilling` typekey indicates that the due date is specified and the bill date is computed from the specified due date.

The upgrade sets `BillDateOrDueDateBilling` to `DueDateBilling` for `Account` rows where `FixDueDayOfMonth` equals 1. For `Account` rows where `FixDueDayOfMonth` does not equal 1, the upgrade sets `BillDateOrDueDateBilling` to `BillDateBilling`.

This step also sets the `InvoiceDayOfMonth` to the `DueInvoiceDayOfMonth` for accounts with due day of month billing (`FixDueDayOfMonth` equals 1).

Finally, the upgrader drops the `DueDayOfMonth` and `FixDueDayOfMonth` columns from `bc_Account`.

This step only runs for upgrades from versions of BillingCenter prior to 3.0.2.

Updating FundsTransfer to Reference Accounts Instead of PolicyPeriods

This step upgrades `bc_FundsTransfer` rows which have a `SourcePolicyPeriod` and `TargetPolicyPeriod` reference to instead populate the `SourceAccount` and `TargetAccount` fields.

The upgrader then drops the `SourcePolicyPeriod` and `TargetPolicyPeriod` columns from `bc_FundsTransfer`.

This step only runs for upgrades from versions of BillingCenter prior to 3.0.2.

Renaming PaymentPlan Column

The upgrade renames the `bc_Plan` column `DaysBeforePlcyExpForInvCutoff` to `DaysBeforePlcyExpForInvBlckout`. This column is used by the `PaymentPlan` subtype. The name of the property outside the database context is `PaymentPlan.DaysBeforePolicyExpirationForInvoicingBlackout`.

BillingCenter 8.0.4 includes an `InvoicingBlackoutType` column on `PaymentPlan`. `InvoicingBlackoutType` is a typelist with types billed and due.

This step applies for upgrades from versions prior to BillingCenter 3.0.2.

Converting Policy-level NegativeWriteoffs to Account-level NegativeWriteoffs

The upgrade converts `NegativeWriteoffs` at the `Policy` level into `NegativeWriteoffs` at the `Account` level, using the following steps:

- Modifying records in table `bc_NegativeWriteoff`:
 - For each record with a non-null `PolicyPeriodID`, populate `AccountID` with `PolicyPeriodID.AccountID`.

- Drop column PolicyPeriodID.
- For each record with a Subtype of PolicyNegativeWriteoff, change the Subtype to AcctNegativeWriteoff.
- Inserting new rows in bc_AccountContext based on relevant rows in bc_PolicyPeriodContext.
 - The subtype will be NegWriteoffAcctContext instead of NegWriteoffPlcyContext.
 - Populate bc_AccountContext.AccountID from PolicyPeriodID.
 - Copy bc_PolicyPeriodContext.PolicyNegativeWriteoffID to bc_AccountContext.AcctNegativeWriteoffID.
 - For each Transaction record with a Subtype of NegativeWriteoffPolicy, change it to NegativeWriteoffAcct.

This step applies for upgrades from versions prior to BillingCenter 3.0.2.

Converting DBNegativeChargeBilled Transactions to ChargeBilled

The upgrade updates bc_Transaction.Subtype to change instances with typecode for DBNegChargeBilled with the typecode for ChargeBilled.

This step applies for upgrades from versions prior to BillingCenter 3.0.2.

Removing Policy Period Unapplied Transactions

For upgrades from versions prior to BillingCenter 3.0.2, the upgrade:

- Changes the Transaction subtype PolicyFundsTransferred to AcctsFundsTransferred.
- Changes the Transaction subtype FundsTransferAcctToPol to AcctsFundsTransferred.
- Changes the Transaction subtype FundsTransferPolToAcct to AcctsFundsTransferred.
- Changes the Transaction subtype PolProdTransfer to AcctProdTransfer.
- Changes the Transaction subtype ProdPolTransfer to ProdAcctTransfer.
- Updates TransferContext.SourceAccountID and TransferContext.TargetAccountID to the account on the source or target policy period.
- Drops columns TransferTxContext.SourcePolicyPeriodID and TransferTxContext.TargetPolicyPeriodID.
- Deletes PaymentDistributed transactions.
- Deletes TransferExcess transactions.
- Deletes UnappliedFundsRollup transactions.
- Deletes ExcessDistributed transactions.
- Deletes NegativeWriteoffRollup transactions and related context records from bc_policyperiodcontext.
- Drops column bc_acctinvcpolpersnapshot.unappliedamount.
- Adds records to TmpPolicyTAccount, to create a mapping of policy TAccounts to account TAccounts for PolicyPeriod unapplied transactions.
- Changes line items for PolicyPeriod unapplied TAccounts so that they point to Account unapplied TAccounts.
- Deletes PolicyPeriod unapplied TAccounts.
- Deletes PolicyPeriod unapplied TAccountPatterns.
- Adds records to TmpPolicyTAccount to create a mapping of policy TAccounts to account TAccounts for PolicyPeriod negative write-off transactions.
- Deletes PolicyPeriod negative write-off TAccounts.
- Deletes PolicyPeriod negative write-off TAccountPatterns.

This step applies for upgrades from versions prior to BillingCenter 3.0.2.

Upgrading Transaction LineItems

This step upgrades `Transaction LineItems`, by doing the following:

- Removes charge reserve CREDIT `LineItem` for `InitialChargeTxn`.
- Adds account unapplied CREDIT `LineItem` for `InitialChargeTxn`.
- Removes account unapplied CREDIT `LineItem` from `ChargePaidFromUnapplied`.
- Removes charge reserve DEBIT `LineItem` from `ChargePaidFromUnapplied`.

This step applies to upgrades from versions prior to 3.0.3.

Cleaning ProcessHistory

The upgrade deletes `ProcessHistory` records for batch process types that no longer exist and are therefore missing from `bct1_ProcessHistoryType`.

This step applies to upgrades from versions prior to 3.0.3.

Renaming ProducerCashToPayableXferTxn to ProducerPayableTransferTxn

BillingCenter 3.0.2 added a transaction called `ProducerCashToPayableXferTxn`. In BillingCenter 3.0.3 this transaction and associated authority limits and transaction contexts was renamed to `ProducerPayableTransferTxn`. The upgrade drops artifacts of the old name. The upgrade drops the `bc_ProducerCashToPayableXfer` and `bcst_ProducerCashToPayableXfer` tables and drops the following columns:

- `bc_Activity.ProducerCashToPayableXferID`
- `bc_AuthorityEvent.ProdPbleToCashXferAuthEventID`
- `bc_ProducerContext.ProducerCashToPayableXferID`
- `bc_ProducerContext.TakesCashFromID`

This step applies for upgrades from versions prior to BillingCenter 3.0.3.

Linking PaymentPlanModifier to First Premium Charge

The upgrade links the `PaymentPlanModifier` to the `ReferenceCharge` which is the first premium charge on the policy period.

This step applies to upgrades from versions prior to 3.0.3.

Changing PaymentPlanModifier Subtype MatchNumberOfPlannedInstallments

The upgrade replaces the `PaymentPlanModifier` subtype `MatchNumberOfPlannedInstallments` with the subtype `MatchPlannedInstallments`.

The old `PaymentPlanModifier` subtype `MatchNumberOfPlannedInstallments` modified a payment plan to change the `maximumNumberOfInstallments` to be the same as the number of planned installments for a given policy period. The new `PaymentPlanModifier` subtype `MatchPlannedInstallments` changes the `maximumNumberOfInstallments` to be the same as the number of planned installments for the issuance charge.

This step applies to upgrades from versions prior to 3.0.3.

Dropping PaymentPlanModifier.PolicyPeriodID

The upgrade drops the `PaymentPlanModifier.PolicyPeriodID` column.

This step applies to upgrades from versions prior to 3.0.3.

Adding or Renaming PaymentPlanID on BillingInstruction

For upgrades from 2.1 versions, the upgrade adds a `PaymentPlanID` column to `bc_BillingInstruction`. For upgrades from 3.0 versions prior to 3.0.3, the upgrade renames the `bc_BillingInstruction` column `ModifiedPaymentPlanID` to `PaymentPlanID`.

Dropping Tables Relevant to Money Received and Payments

For upgrades from versions prior to BillingCenter 3.0.3, the upgrade drops the tables relevant to money received and payments, including:

- `bc_payment`
- `bc_paymentcontext`
- `bc_chargepaidsource`
- `bc_policyperiodcontext`
- `bctl_policyperiodcontext`
- `bcst_policyperiodcontext`
- `bctt_policyperiodcontext`
- `bc_suspendeitemcontext`
- `bcst_suspendeitemcontext`
- `bctt_suspendeitemcontext`
- `bc_basesuspdistitem`
- `bcst_basesuspdistitem`
- `bctt_basesuspdistitem`
- `bcst_prodsuspensecontext`
- `bctt_transferexcesstype`
- `bcst_producercoderoleentry`
- `bc_modifiedpayment`

This step applies to upgrades from versions prior to 3.0.3.

Dropping CollateralContext Columns

The upgrade drops the columns `bc_CollateralContext.PaymentID` and `bc_CollateralContext.PolicyPeriodID`.

This step applies to upgrades from versions prior to 3.0.3.

Populating Writeoff.ReversedAmount

The upgrade populates `Writeoff.ReversedAmount` with the sum of its reversed `ChargeWrittenOff` transactions respectively.

This step applies to upgrades from versions prior to 3.0.4.

Dropping TmpDistItem.MoneyReceivedID

The upgrade drops the `PaymentPlanModifier.PolicyPeriodID` column.

This step applies to upgrades from versions prior to 3.0.4.

Renaming the ProducerCodeRoleEntry Table

The upgrade renames the table for the `ProducerCodeRoleEntry` from `bc_ProducerCodeRoleEntry` to `bc_ProdCodeRoleEntry`.

This step applies to upgrades from versions prior to 3.0.4.

Setting HiddenOverridingDefaultPayer on Charge to a ProducerCode

The upgrade sets `bc_Charge.HiddenOverridingDefaultPayer` to the corresponding `ProducerCode` if the `HiddenOverridingDefaultPayer` is set to a Producer.

This step applies to upgrades from versions prior to 3.0.4.

Resetting ChargeProRataTx Work Item Table

To accommodate data model changes to the `bc_ChargeProRataTxWorkItem` table, the upgrade deletes all references to rows in the work item table. These references include process history, instrumented worker, and instrumented worker task. The upgrade then drops the table. The table is later regenerated with the new data model.

This step applies to upgrades from versions prior to 3.0.4.

Resetting Work Item Tables

To accommodate data model changes to the following work item tables, the upgrade deletes all references to rows in the work item table. These references include process history, instrumented worker, and instrumented worker task. The upgrade then drops the work item tables. The tables are later regenerated with the new data model.

The following tables are reset for upgrades from versions prior to 3.0.5.

Table	Associated process
<code>bc_CmsnPayableWorkItem</code>	<code>CmsnPayable</code>
<code>bc_InvoiceBilledWorkItem</code>	<code>Invoice</code>
<code>bc_InvoiceDueWorkItem</code>	<code>InvoiceDue</code>

This step applies to upgrades from versions prior to 3.0.5.

Consolidating Duplicate Policy Commissions

Prior to BillingCenter 3.0.6, duplicate policy commissions could be created. The upgrade identifies and removes the duplicates. A `PolicyCommission` is considered a duplicate if its `PolicyPeriod`, `Role`, and `ProducerCode` are not unique. The upgrade also deletes the related `TAccountContainer` entities and their related `TAccount` entities. The upgrade updates all foreign keys to the deleted items to point to the original items.

The upgrade performs the following actions:

- Populates `bc_TmpPolicyCommissionDup` with `PolicyCommission` duplicates.
- Updates `bc_TmpPolicyCommissionDup` so that duplicate charge commissions have a reference to the charge commission being retained.
- Populates `bc_TmpChargeCommissionDup` with potentially duplicate charge commissions. Potentially duplicate charge commissions are charge commissions related to duplicate policy commissions.
- Populates `bc_TmpChargeCommissionUnique` with one row for each charge commission being kept.
- Updates `bc_TmpChargeCommissionDup` table so that duplicate charge commissions have a reference to the charge commission being retained.
- Inserts potential duplicate item commission records into a temporary table.
- Populates `bc_TmpItemCommissionUnique` with one row for each item commission being kept.
- Updates `bc_TmpItemCommissionDup` so that duplicate item commissions have a reference to the item commission being retained.
- Inserts potential duplicate t-account records into a temporary table.
- Updates `bc_TmpTaccountDup` with the identifier of the replacement t-account for the t-accounts to be deleted.

- Updates foreign key references to deleted item commissions.
- Deletes duplicate item commissions.
- Updates the Active field on bc_ItemCommission to account for deleted duplicate item commissions.
- Updates foreign key references to deleted item commissions.
- Updates foreign key references to deleted charge commissions.
- Consolidates duplicate entries in the bc_OrigChCmsn edge table. Duplicate entries have the same OwnerID after consolidation of charge commissions.
- Deletes duplicate charge commissions.
- Updates foreign key references to deleted policy commissions.
- Deletes duplicate policy commissions.
- Updates defaultForPolicy on bc_PolicyCommission to account for deleted duplicate policy commissions.
- Updates foreign key references to deleted t-accounts.
- Deletes duplicate t-accounts.
- Updates foreign key references to deleted t-account containers.
- Deletes duplicate t-account containers.
- Updates denormalized transfer fields PrimaryPolicyPeriodID, DefaultForPolicy, StartDate, and EndDate on bc_PolicyCommission to reflect consolidation.
- Updates Active status on bc_ItemCommission after bc_PolicyCommission.defaultforPolicy has been updated.

This step applies to upgrades from versions prior to 3.0.6.

Erasing Database-based Archiving

The upgrade removes tables and columns used for archiving from the database. The upgrade drops the following tables and columns:

- bc_ArchiveAdminKey
- bc_ArchiveGraphRecord
- bc_ArchiveTransitionRec
- bc_ArchiveTypeKey

The database archiving feature was not available for BillingCenter.

Removing InetSoft Reporting Support

The upgrade removes the following database elements that were involved in supporting InetSoft reporting:

- bc_reportgroup
- bc_rolerptprivilege
- bc_rptgroup rpt
- bc_sreereport

The upgrade also drops the bc_privilege table, which is rebuilt later in the upgrade. Finally, the database upgrade removes the reporting_admin permission.

Adding NotificationSent to ProcessHistory

The upgrade creates a NotificationSent bit column on ProcessHistory. The upgrade sets the default value of ProcessHistory.NotificationSent for existing rows to true, so BillingCenter does not resend notifications.

Setting Parameter for Data Files Imported

The upgrade sets the `data_files_imported` parameter to `finished` in the `bc_Parameter` table, if the parameter is not already listed. This prevents rare issues with the upgrade caused by dependency on this parameter.

Dropping Extractable Columns

The upgrade removes the following columns from each entity that implements the `Extractable` delegate:

- `archiveid`
- `archivepartition`
- `extractready`
- `partition`

Not every `Extractable` entity includes these columns. The upgrade drops any of these columns that do exist on an `Extractable` entity.

This step is part of the removal of database-based archiving. The database archiving feature was not available for BillingCenter.

Setting IndividualStacks column on WorkQueueProfilerConfig to Non-nullable

The upgrade sets the `IndividualStacks` column on `bc_WorkQueueProfilerConfig` to non-nullable.

Changing Contact Foreign Keys on ContactAutoSyncWorkItem

This is a Guidewire platform-level upgrade step. Because BillingCenter does not use the Contact Auto Sync work queue, this step does not affect BillingCenter.

The upgrade first checks that the `bc_ContactAutoSyncWorkItem` table is empty. Then the upgrade drops the `minContactID` and `maxContactID` foreign keys to `bc_Contact` and replaces them with soft entity references `minContactRef` and `maxContactRef`.

Checking for Null Effective-dated Foreign Keys on EffDatedOnly Delegates

The upgrade checks that no effective-dated foreign keys on `effDatedOnly` delegates are null. If the upgrade finds any null effective-dated foreign keys on `effDatedOnly` delegates, it reports an error and stops the upgrade. The error includes an SQL query to identify the rows with issues.

Adding Subtype to WorkflowWorkItem

The upgrade adds a `Subtype` column to `cc_WorkflowWorkItem` with a default value of `WorkflowWorkItem`.

Renaming Primary Key Constraints and Indexes to Indicate Table Name

The upgrade renames primary key constraints and indexes to indicate the table name. For example, on Oracle, the upgrade renames the primary key index on `bc_Activity` to `PK_Activity`. On SQL Server, the upgrade renames the primary key index on `bc_Activity` to `bc_Activity_PK`.

Updating Columns to Support Very Large Data Sets

The upgrade changes the datatype of some columns in tables related to data distribution, data loading, and table statistics to be able to support very large data sets. In particular, on SQL Server the upgrade changes INT columns to BIGINT. For Oracle and SQL Server the upgrade changes DECIMAL columns to BIGINT for cases in which the column holds whole numbers.

This affects the following tables:

- `bc_ArrayDataDist`

- bc_ArraySizeCntDD
- bc_AssignableForKeyDataDist
- bc_AssignableForKeySizeCntDD
- bc_BeanVersionDataDist
- bc_BlobColDataDist
- bc_BooleanColDataDist
- bc_ClobColDataDist
- bc_DateAnalysisDataDist
- bc_DateBinnedDDDateBin
- bc_DateBinnedDDValue
- bc_ForKeyDataDist
- bc_GenericGroupCountDataDist
- bc_HourAnalysisDataDist
- bc_LoadInsertSelect
- bc_LoadOperation
- bc_LoadRowCount
- bc_NullableColumnDataDist
- bc_TableDataDist
- bc_TableUpdateStats
- bc_TypecodeCountDataDist
- bc_TypekeyDataDist

Dropping Staging Tables

The database upgrade drops the following staging tables:

- bcst_TmpABCommission
- bcst_TmpChargeCmsnWriteoff
- bcst_TmpCPPF
- bcst_TmpInvStreamCreationOrder
- bcst_TmpItemStateCmsn
- bcst_TmpPayment
- bcst_TmpPolicy

The upgrade also drops the LoadCommandID column from the production table for each of these entities. These entities are not loadable as of BillingCenter 7.0.

This step applies to upgrades from versions prior to 7.0.1.

Converting Item Events from Retireable to Editable

The database upgrade checks that there are no retired `ItemEvent` records in `bc_ItemEvent`. If there are no retired `ItemEvent` records, the upgrade deletes the `bc_ItemEvent.Retired` column. If the upgrade detects retired `ItemEvent` records, it reports an error and stops the upgrade.

This step applies to upgrades from versions prior to 7.0.1.

Converting Line Items from Retireable to Editable

The database upgrade checks that there are no retired `LineItem` records in `bc_LineItem`. If there are no retired `LineItem` records, the upgrade deletes the `bc_LineItem.Retired` column. If the upgrade detects retired `LineItem` records, it reports the error and stops the upgrade.

This step applies to upgrades from versions prior to 7.0.1.

Adding ID Suffix to Delinquency Processing Columns

The database upgrade adds the suffix `ID` to the `LastDelinquencyProcessGrp` and `LastDelinquencyProcessUser` columns of the `bc_dynamic_assign` and `bc_group_assign` tables if it is not already present.

This step applies to upgrades from versions prior to 7.0.1.

Regenerating Work Item Tables

The database upgrade deletes all references to rows in the work item tables, including `bc_ProcessHistory`, `bc_InstrumentedWorker`, `bc_InstrumentedWorkerTask`, and then drops the tables. The tables are regenerated when BillingCenter starts and include any data model changes that Guidewire made to the work item tables between versions.

Populating Charge.WrittenDate and PolicyPeriod.TermConfirmed

The database upgrade adds the columns `bc_PolicyPeriod.TermConfirmed` and `bc_Charge.WrittenDate`. The upgrade sets `bc_PolicyPeriod.TermConfirmed` to `true`.

If the Charge belongs to a `BillingInstruction` with Subtype of `Audit`, `Cancellation`, `PolicyChange`, `PremiumReportBI`, or `Reinstatement`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `BillingInstruction` has a more recent `ModificationDate`. If the `BillingInstruction` has a `ModificationDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `ModificationDate` of the `BillingInstruction`.

If the Charge belongs to a `BillingInstruction` with Subtype of `BaseGeneral`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `PolicyPeriod` of the `BillingInstruction` has a more recent `PolicyPerEffDate`. If the `PolicyPeriod` of the `BillingInstruction` has a `PolicyPerEffDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `PolicyPerEffDate` of the `PolicyPeriod`.

If the Charge belongs to a `BillingInstruction` with Subtype of `Issuance`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `IssuancePolicyPeriod` of the `BillingInstruction` has a more recent `PolicyPerEffDate`. If the `IssuancePolicyPeriod` of the `BillingInstruction` has a `PolicyPerEffDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `PolicyPerEffDate` of the `IssuancePolicyPeriod`.

If the Charge belongs to a `BillingInstruction` with Subtype of `NewRenewal`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `NewRenewalPolicyPeriod` of the `BillingInstruction` has a more recent `PolicyPerEffDate`. If the `NewRenewalPolicyPeriod` of the `BillingInstruction` has a `PolicyPerEffDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `PolicyPerEffDate` of the `NewRenewalPolicyPeriod`.

If the Charge belongs to a `BillingInstruction` with Subtype of `Renewal`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `RenewalPolicyPeriod` of the `BillingInstruction` has a more recent `PolicyPerEffDate`. If the `RenewalPolicyPeriod` of the `BillingInstruction` has a `PolicyPerEffDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `PolicyPerEffDate` of the `RenewalPolicyPeriod`.

If the Charge belongs to a `BillingInstruction` with Subtype of `Rewrite`, the upgrade sets `bc_Charge.WrittenDate` to `bc_Charge.ChargeDate` unless the `RewritePolicyPeriod` of the `BillingInstruction` has a more recent `PolicyPerEffDate`. If the `RewritePolicyPeriod` of the `BillingInstruction` has a `PolicyPerEffDate` more recent than `bc_Charge.ChargeDate`, the upgrade sets `bc_Charge.WrittenDate` to the `PolicyPerEffDate` of the `RewritePolicyPeriod`.

If the Charge belongs to a BillingInstruction with Subtype of PremiumReportDueDate, the upgrade sets bc_Charge.WrittenDate to bc_Charge.ChargeDate unless the PremiumReportDDPolicyPeriod of the BillingInstruction has a more recent PolicyPerEffDate. If the PremiumReportDDPolicyPeriod of the BillingInstruction has a PolicyPerEffDate more recent than bc_Charge.ChargeDate, the upgrade sets bc_Charge.WrittenDate to the PolicyPerEffDate of the PremiumReportDDPolicyPeriod.

If the Charge belongs to a BillingInstruction with Subtype of AccountGeneral, CollateralBI, or SegregatedCollReqBI, the upgrade sets bc_Charge.WrittenDate to bc_Charge.ChargeDate.

This step applies to upgrades from versions prior to 7.0.1.

Resetting Upgrade Commission Work Item Table

The upgrade deletes bc_UpgradeCommissionWorkItem, the work item table for the Upgrade Commission work queue. The server rebuilds the bc_UpgradeCommissionWorkItem table afterwards.

This step applies to upgrades from versions prior to 7.0.1.

Dropping Temporary Tables Used for Previous Upgrades

The upgrade drops the following temporary tables and any corresponding staging tables, if they exist:

- FKTempTable
- NewIDsTempTable
- TmpChargeAcctGeneral
- TmpChargeCmsnWriteoff
- TmpChargeCommissionDup
- TmpChargeCommissionUnique
- TmpChargeGeneral
- TmpCmsnWrtffDlt
- TmpInvStreamCreationOrder
- TmpItemCommissionDup
- TmpItemCommissionUnique
- TmpOrigChargeCmsnConsol
- TmpPolicyCommissionDup
- TmpPolicyCommissionUnique
- TmpTAccountDup

These tables were used in previous upgrades and are no longer needed.

This step applies to upgrades from versions prior to 7.0.2.

Resetting Invoice Work Item Table

The upgrade deletes the bc_InvoiceBilledWorkItem work item table for the Invoice work queue. The server rebuilds the bc_InvoiceBilledWorkItem table afterwards.

This step applies to upgrades from versions prior to 7.0.2.

Resetting Invoice Due Work Item Table

The upgrade deletes the bc_InvoiceDueWorkItem work item table for the Invoice Due work queue. The server rebuilds the bc_InvoiceDueWorkItem table afterwards.

This step applies to upgrades from versions prior to 7.0.2.

Dropping PaymentComments from TmpDistItem

The upgrade drops the `bc_TmpDistItem.PaymentComments` column if it exists.

This step applies to upgrades from 7.0 versions prior to 7.0.3.

Populating Null PayableCriteria Columns on ChargeCommission

The upgrade populates any null `PayableCriteria` columns on `ChargeCommission` with the `PayableCriteria` value of the parent `CommissionSubPlan`.

This step applies to upgrades from versions prior to 7.0.3.

Populating Null PayableCriteria Columns on ItemCommission

The upgrade populates any null `PayableCriteria` columns on `ItemCommission` with the `PayableCriteria` value of the parent `ChargeCommission`.

This step applies to upgrades from versions prior to 7.0.3.

Converting Legacy CmsnTransferFromRollup Transactions

The upgrade removes any `CmsnTransferFromRollup` transactions that were created in BillingCenter 1.0, and replaces them with `ReserveCmsnEarned` and `PolicyCmsnPayable` transactions. The upgrade creates `ReserveCmsnEarned` and `PolicyCmsnPayable` transactions at the `ChargeCommission` level to duplicate the line items of `CmsnTransferFromRollup` at the `PolicyCommission` level. You could not create `CmsnTransferFromRollup` transactions in BillingCenter 2.0 or higher. So this step only does anything for databases that have previously been upgraded from BillingCenter 1.0 to 2.1 or 3.0 and still contain `CmsnTransferFromRollup` transactions.

The upgrade also removes the `CmsnTransferFromRollup` subtype from `Transaction`.

Before converting the transactions, the upgrade checks that there are no `CmsnTransferFromRollup` transactions that are reversed.

This step applies to upgrades from versions prior to 7.0.3.

Removing Legacy NegCmsnWriteoff Transactions

This step removes the legacy `NegCmsnWriteoff` transaction subtype. This transaction subtype has been deprecated since BillingCenter 2.0.

This step applies to upgrades from versions prior to 7.0.3.

Removing Legacy TimeBasedIncEarned Transactions

This step first checks that there are no legacy transactions of subtype `TimeBasedIncEarned`. If the upgrade does not detect such transactions, it removes the `TimeBasedIncEarned` transaction subtype. If the upgrade does detect such transactions, it reports an error and stops the upgrade.

This step applies to upgrades from versions prior to 7.0.3.

Dropping ItemEventCount from InvoiceItem

The upgrade drops the `bc_InvoiceItem.ItemEventCount` column.

This step applies to upgrades from versions prior to 7.0.3.

Adding and Populating AllInvoiceItemsExactlyPaid Bit Column to Invoice

The upgrade adds an `AllInvoiceItemsExactlyPaid` column to `bc_Invoice`. The upgrade sets `AllInvoiceItemsExactlyPaid` to true for an Invoice if all invoice items are exactly paid. The following conditions must apply:

- The Invoice subtype is `StatementInvoice`.
- The Invoice `NetAmount` minus `NetAmountPaid` minus `NetAmountWrittenOff` equals 0.
- For all `InvoiceItem` records for the Invoice, the `PrimaryCommissionAmount` minus `PrimaryPaidCommission` minus `PrimaryCmsnPayableAmount` minus `PrimaryWrittenOffCommission` equals 0.

This step applies to upgrades from versions prior to 7.0.4.

Adding GrossSettled Denormalized Bit Field To InvoiceItem

The upgrade adds a `GrossSettled` denormalized bit field to `bc_InvoiceItem` and sets the value of `GrossSettled` to true if `Amount` is equal to `PaidAmount` plus `GrossAmountWrittenOff`.

This step applies to upgrades from versions prior to 7.0.4.

Converting T-account Table from Retireable to Editable

The upgrade checks that there are no retired `TAccount` records in `bc_TAccount`. If there are no retired `TAccount` records, the upgrade deletes the `bc_TAccount.Retired` column. If the upgrade detects retired `TAccount` records, it reports an error and stops the upgrade.

This step applies to upgrades from versions prior to 7.0.4.

Resetting Work Item Tables

For upgrades from 7.0.1 through 7.0.3, the database upgrade drops and recreates the work item tables for the following work queues:

- Full Pay Discount
- Invoice Billed
- Invoice Due
- Payment Request

Resetting Funds Allotment Work Item Table

The upgrade drops and recreates the work item table for the Funds Allotment work queue.

This step applies to upgrades from versions prior to 7.0.4.

Adding FullyAllotted Bit Column to FundsTracker

The upgrade adds a denormalized bit column, `FullyAllotted` to `bc_FundsTracker`. The upgrade sets `FullyAllotted` to true if `AmountAllottedDenorm` equals `TotalAmount`.

This step applies to upgrades from versions prior to 7.0.4.

Adding HasChargeBilledTransaction and HasChargeDueTransaction to InvoiceItem

The upgrade adds `HasChargeBilledTransaction` and `HasChargeDueTransaction` columns to `bc_InvoiceItem`. The upgrade sets `HasChargeBilledTransaction` to true if a `ChargeBilled` exists for the `InvoiceItem`. The upgrade sets `HasChargeDueTransaction` to true if a `ChargeDue` transaction exists for the `InvoiceItem`.

This step applies to upgrades from versions prior to 7.0.4.

Updating Negative Write-off Reversals to be Loadable

The upgrade adds LoadCommandID and ReversalDate columns to bc_NegativeWriteoffRev. Both columns are created with null values. Negative write-off reversals can now be loaded using the bcst_NegativeWriteoffRev staging table.

Updating Negative Write-offs to be Loadable

The upgrade adds a LoadCommandID column to bc_NegativeWriteoff. The LoadCommandID column is created with a null value. Negative write-offs can now be loaded using the bcst_NegativeWriteoff staging table.

Updating T-account Balances

The upgrade updates TAccount balances, using the following steps:

- Creates a TmpTAccount temporary table to use during BalanceDenorm update of all TAccounts.
- Calculates the TAccount.BalanceDenorm field for each TAccount and inserts the value into TmpTAccount.
- Updates the TAccount.BalanceDenorm field for each TAccount using the value in TmpTAccount.

This step applies to upgrades from versions prior to 7.0.4.

Updating Charge TAccountContainer Balances

The upgrade recalculates the BalanceDenorm fields on all charge TAccountContainers.

This step applies to upgrades from versions prior to 7.0.4.

Updating InvoiceItem.PrimaryCommissionAmount

The upgrade updates the bc_InvoiceItem.PrimaryCommissionAmount denormalized field.

This step only runs for upgrades from versions prior to 3.0.6. This step may take a long time to run. If you are upgrading from a version prior to 3.0.8, Guidewire recommends that you first upgrade to 3.0.8 and run consistency checks before continuing the upgrade to 8.0. A direct upgrade from a 3.0 version prior to 3.0.8 can perform poorly, so for the best upgrade performance, upgrade to 3.0.8 first.

Regenerating InstrumentedWorker and Dropping InstrumentedWorkerTask

The database upgrade drops the bc_InstrumentedWorker and bc_InstrumentedWorkerTask tables. The bc_InstrumentedWorker table is regenerated when BillingCenter starts and includes data model changes that Guidewire made to the table between versions. The bc_InstrumentedWorkerTask table replaces the bc_InstrumentedWorkerTask table used in versions prior to BillingCenter 8.0.

Checking Uniqueness of Localized Admin Data

The upgrade checks that the Name value of the following entities is unique for that entity type:

- AuthorityLimitProfile
- BusinessWeek
- Plan
- Region
- Role

Dropping ClusterInfo

The upgrade drops the `bc_ClusterInfo` table. This table is renamed `bc_BatchServer`. The new table is created following the upgrade when BillingCenter starts. The `bc_BatchServer` table always contains only one row that describes the current batch server. This table is used by all cluster nodes to get the address of the current batch server and enforce that only a single batch server exists within the cluster.

BillingCenter 8.0 adds a `ClusterMemberData` entity that contains information about current cluster members. The information from this table is shown on the `Cluster Info` page. JGroups uses this table for the following:

- **JGroups over UDP** – Reporting and audit purposes. UDP multicast is used for discovery.
- **JGroups over TCP** – Reporting and discovery. JGroups reads the IP addresses of the current members from the `bc_ClusterMemberData` table.

Updating SpatialPoint on Address

The upgrade updates the `SpatialPoint` column on `bc_Address` with the data in the `Longitude` and `Latitude` columns. The upgrade checks for rows where `Longitude` or `Latitude` are non-null and the other is null. If the upgrade detects such rows, it reports an error, stops the upgrade, and provides an SQL query to find these rows.

After the upgrade updates `SpatialPoint`, it drops the `HTMID` column and the `Longitude` and `Latitude` columns.

Dropping Foreign Keys to ProcessHistory

The upgrade drops all `ProcessHistoryID` foreign keys that refer to `bc_ProcessHistory`.

Dropping WorkQueueName Column from WorkQueueWorkerControl

The upgrade drops the `bc_WorkQueueWorkerControl.WorkQueueName` column and deletes all current `WorkQueueWorkerControl` records. Later, the upgrade adds a new `LockName` column with unique index records.

Renaming WorkItem Column NumRetries to Attempts

The upgrade renames the `WorkItem` column `NumRetries` to `Attempts`.

Adding Subtype Column to Activity, Address, and History Tables

The upgrade adds a `Subtype` column to the `bc_Activity`, `bc_Address`, and `bc_History` tables. This allows Guidewire applications to create subtypes of these entities as needed.

Upgrading Consistency Check Tables

The upgrade makes the following changes to tables involved in consistency checks:

- drops the `NumThreads` and `Subtype` columns from `bc_dbConsistCheckRun`.
- drops the `Subtype` column from `bc_dbConsistCheckQueryExec`.
- updates null values of the `TableName` and `ThreadName` columns of `bc_dbConsistCheckQueryExec` to the value `UNKNOWN`.

Upgrading Database Statistics Tables

The upgrade makes the following changes to tables involved in gathering database statistics for BillingCenter:

- deletes all `bc_ProcessHistory` records for the Incremental Database Statistics process.
- drops the `Subtype` column from `bc_DatabaseUpdateStats`, `bc_TableUpdateStats`, and `bc_TableUpdateStatsStatement`.
- sets null values for `bc_TableUpdateStatsStatement.ObjectName` and `bc_TableUpdateStatsStatement.UpdateStatsStatement` to `UNKNOWN`.

- drops `bc_DatabaseUpdateStats.NumThreads`.
- drops `bc_TableUpdateStats.Deletes`, `bc_TableUpdateStats.Inserts` and `bc_TableUpdateStats.RowCnt`.

Dropping Upgrade-related Tables

The upgrade drops the following tables:

- `bc_UpgradeDBParameterPair`
- `bc_UpgradeDBParameterRow`
- `bc_UpgradeDBParameterSet`

Dropping AddressBookFingerprint from Contact and ContactCategoryScore

The upgrade drops the `AddressBookFingerprint` column from `bc_Contact` and `bc_ContactCatsScore`. The upgrade also drops the `AddressBookFingerprint` property from the `Contact` and `ContactCategoryScore` entities.

Dropping Obsolete Temporary Upgrade Tables

The upgrade drops the following obsolete temporary upgrade tables:

- `bc_TmpABCommission`
- `bc_TmpChargeAcctGeneral`
- `bc_TmpChargeGeneral`
- `bc_TmpChargePaidTxn`
- `bc_TmpCmsnReductionInvItem`
- `bc_TmpCPFP`
- `bc_TmpDist`
- `bc_TmpDistItem`
- `bc_TmpItemStateCmsn`
- `bc_TmpLineItem`
- `bc_TmpMoneyRcvd`
- `bc_TmpNegBldRevTrans`
- `bc_TmpPayment`
- `bc_TmpPolicy`
- `bc_UpgradeCollReq`

Removing Upgrade Commission Batch Process

The upgrade drops the `bc_UpgradeCommissionWorkItem` table as part of removing the Upgrade Commission batch process.

Dropping Columns

The upgrade drops the following columns:

- `bc_Charge.LegacyDepositOverride`
- `bc_InvoiceStream.Name`
- `bc_PolicyPeriod.ModNumber`

Updating Permissions

The upgrade updates permissions during the upgrade.

The upgrade converts some `feecreate` privileges to `gentxn` before removing remaining `feecreate` privileges. The upgrade updates `bc_Privilege` to update `Permission` from `feecreate` to `gentxn` if the `RoleID` is not associated with another privilege with permission `gentxn`.

The upgrade removes the following privileges:

- `acctdistedit`
- `acctdistview`
- `feecreate`

The upgrade adds the following privileges:

Privilege	Roles
<code>admindatachangeview</code>	<code>superuser</code>
<code>admindatachangeexec</code>	<code>superuser</code>
<code>wsdatachangeedit</code>	<code>superuser</code>
<code>retremplancreate</code>	<code>general_admin</code> <code>plan_admin</code> <code>superuser</code>
<code>retremplanedit</code>	<code>general_admin</code> <code>plan_admin</code> <code>superuser</code>
<code>retremplanview</code>	<code>billing_clerical</code> <code>billing_manager</code> <code>commissions_admin</code> <code>finance_manager</code> <code>general_admin</code> <code>plan_admin</code> <code>superuser</code> <code>underwriter</code>
<code>prodpmntedit</code>	<code>general_admin</code> <code>superuser</code>
<code>prodpromedit</code>	<code>general_admin</code> <code>superuser</code>
<code>prodpromview</code>	<code>general_admin</code> <code>superuser</code>

Renaming Class Names of Transactions to Remove Abbreviations

The upgrade modifies class names of transactions to remove abbreviations, except for the word 'Transaction' itself.

Old Typekey	New Typekey
<code>AgencyMoneyReceivedTxn</code>	<code>AgencyBillMoneyReceivedTxn</code>
<code>DirectBillMoneyRcvdTxn</code>	<code>DirectBillMoneyReceivedTxn</code>

Old Typekey	New Typekey
NegativeWriteoffAcct	AccountNegativeWriteoffTxn
CommissionReserved	CommissionsReserveTxn
CmsnReductionTxn	CommissionsReserveWriteoffTxn
CmsnPayableReduction	CommissionsReserveNegativeWriteoffTxn
CmsnRcvableReduction	CommissionsReservePositiveWriteoffTxn
NegWriteoffAcctContext	AccountNegativeWriteoffContext

The AccountNegativeWriteoffContext typekey is on bctl_AccountContext. The rest of the typekeys are on bctl_Transaction.

Adding Currency Columns

The upgrade adds silo currency columns to each siloed table and initializes these currency columns with the system default currency.

Upgrading CommissionWriteoffDistItem

The upgrade updates the class name, column names, and subtypes of the `CommissionWriteoffDistItem` entity. The upgrade makes the following changes:

- Renames the `CmsnReductionID` column of `bc_ProducerContext` to `CommissionWriteoffDistItemID`.
- Renames the `bctl_ProducerContext` typekey `CmsnReductionContext` to `CommissionWriteoffDistItemContext`.
- Renames the `bctl_CmsnReductionType` typekey `payable` to `negative`.
- Renames the `bctl_CmsnReductionType` typekey `receivable` to `positive`.
- Renames the `bc_CmsnReduction` column `ReductionDate` to `DateWrittenOff`.

Updating Subtypes on all ProducerPaymentSent Transaction Contexts

BillingCenter 7.0.4 adds a `ProdPymtSentContext` subtype on `ProducerContext` for `ProducerPaymentSent` transactions. This trigger updates the `ProducerContext` associated with all `ProducerPaymentSent` transactions to `ProdPymtSentContext`.

Converting Tables from Retireable to Editable

The upgrade converts the following tables from `Retireable` to `Editable`.

- `bc_AccountContext`
- `bc_AgencyMoneyRcvdContext`
- `bc_AgencyDisbPaidContext`
- `bc_BillingInstruction`
- `bc_Charge`
- `bc_ChargePatternContext`
- `bc_ChargeProRataTx`
- `bc_CmsnsExpenseRollupCtx`
- `bc_CollateralContext`
- `bc_CreditContext`
- `bc_DBMoneyRcvdContext`
- `bc_NonReceivableItemCtx`

- bc_SuspPymtContext
- bc_TAccountContainer
- bc_TransferTxContext

Ensuring No Retired Rows in Certain Tables

The upgrade checks that there are no retired rows in the bc_ChargeInstanceContext, bc_ProducerContext, or bc_Transaction tables. If the upgrade detects any retired rows in these tables, it reports an error and stops the upgrade.

Removing Unused Columns from TAccount

The upgrade removes the unused Description column from bc_TAccount. The upgrade also removes the Retired column, converting TAccount to an editable entity. The upgrade first checks that there are no retired rows in bc_TAccount. If the upgrade detects any retired bc_TAccount rows, it reports an error and stops the upgrade.

Updating Invoice Item for Exception Status Locking

The upgrade renames fields related to carrying forward exceptions on invoice items in 7.0 to the 8.0 exception lock versions.

Renames bc_InvoiceItem column PaymentCarriedForwardDate to PaymentExceptionLockDate.

Adds a PaymentExceptionLock typekey column to bc_InvoiceItem. The upgrade initializes PaymentExceptionLock to none. If the PaymentExceptionLockDate (formerly PaymentCarriedForwardDate) is not null, the upgrade sets PaymentExceptionLock to notexception.

Renames bc_InvoiceItem column PromiseCarriedForwardDate to PromiseExceptionLockDate.

Adds a PromiseExceptionLock typekey column to bc_InvoiceItem. The upgrade initializes PromiseExceptionLock to none. If the PromiseExceptionLockDate (formerly PromiseCarriedForwardDate) is not null, the upgrade sets PromiseExceptionLock to notexception.

Renaming BaseDist Column NetAmountDistributed to NetDistributedToInvoiceItems

The upgrade renames the bc_BaseDist column NetAmountDistributed to NetDistributedToInvoiceItems.

Updating BillingCenter for Change from State to Jurisdiction

All Guidewire InsuranceSuite applications have been updated to use Jurisdiction instead of State. The upgrade renames:

- PolicyPeriod.RiskState to RiskJurisdiction
- CondCmsnSubPlan.AllStates to AllJurisdictions
- CondCmsnSubPlanState.State to Jurisdiction
- bc_CondCmsnSubplanState to bc_CondCmsnSubplanJurisdiction

Checking References to Jurisdiction Typelist

The upgrade checks that the following references to the Jurisdiction typelist are valid:

- bc_CommissionSubplan.AllJurisdictions
- bc_CondCmsnSubplanState.Jurisdiction
- bc_PolicyPeriod.RiskJurisdiction

Moving AppliedDate to BaseMoneyReceived

The upgrade moves `AppliedDate` from `PaymentMoneyReceived` to `BaseMoneyReceived` and updates all promises that did not used to have the applied date to set their `AppliedDate` to the `ReceivedDate`.

Checking for Promises Linked to ProducerWriteoffs

The upgrade checks that there are no `Promises` linked to `ProducerWriteoffs`. If the upgrade detects `Promises` linked to `ProducerWriteoffs`, it reports an error and stops the upgrade.

Checking Payment Instruments of Agency Bill Payments

The upgrade checks that all agency bill payments with a zero amount have the producer unapplied payment instrument. The upgrade also checks that all agency bill payments with a non-zero amount do not have the producer unapplied payment instrument. If either condition is detected, the upgrade reports an error and stops.

Checking Payment Instruments of Direct Bill Payments

The upgrade checks that all direct bill payments with a zero amount have the account unapplied payment instrument. The upgrade also checks that all direct bill payments with a non-zero amount do not have the account unapplied payment instrument. If either condition is detected, the upgrade reports an error and stops.

Renaming AgencyCycleDist to AgencyCyclePaymentID on ProdWriteoffContext

The upgrade renames the `AgencyCycleDistID` column to `AgencyCyclePaymentID` on `bc_ProducerContext`.

Upgrading PromisedMoney

The upgrade moves the `MoneyBeingModified` edge link from the subclass `PaymentMoneyReceived` to the superclass `BaseMoneyReceived`. For each `PromisedMoney` marked as `Modified`, the upgrade attempts to determine which `PromisedMoney` it was modified into by finding a more recently created `PromisedMoney` attached to the same distribution. The upgrade then creates a link to the more recently created `PromisedMoney` in the `MoneyBeingModified` edge link table. This makes it so that promised monies are attached to the money they are modifying, just like any other money.

Upgrade the Distributed Amounts on Applied Promises

Trigger to upgrade the distributed amount on applied promises. In versions prior to 8.0, when a promise was applied, BillingCenter reversed the distributed amount down to zero. In 8.0, when viewing a promise that has been applied, you can see how much was distributed on the promise even if those amounts have been applied towards payment items.

Checking Amounts on Invoice Items

The upgrade checks that there are no positive amounts applied to negative invoice items or negative amounts applied to positive invoice items. The `netAmountOwed` and the `netAmountToApply` of a distribution must both be positive or both be negative. Similarly, the `CommissionAmountToApply` and `PrimaryCommissionAmount` must both be positive or both be negative. If the upgrade detects a distribution that does not meet this condition, it reports an error and stops.

Upgrading BillingLevel Typelist Values on Account

The upgrade sets all legacy `bc_Account` records that had `BillingLevel` set to `PolicyLevelBilling` to have `BillingLevel` of `PolicyDefaultUnapplied`. The upgrade sets all legacy `bc_Account` records that had `BillingLevel` set to `AccountBilling` to have `BillingLevel` of `Account`.

Wrapping Default Unapplied T-Accounts in UnappliedFunds

The upgrade wraps all default unapplied T-accounts with a new `UnappliedFund` entity. BillingCenter 8.0 uses the `UnappliedFund` entity to contain default unapplied T-accounts to:

- Provide database level protection. Queries can be against the `UnappliedFund` wrapper.
- Validate and protect at compile time.
- Prevent passing around of raw T-accounts with the potential of misuse.
- Provide a container for foreign keys to policy, reporting group, and others.
- Provide a container for methods like funds tracking methods and others.
- Make staging easier.
- Provide an entity on which users can put extension fields to enable linking the unapplied T-account to something other than a policy.

Making Work Item Tables Not Loadable

The upgrade drops the staging tables `bcst_AdvanceExpiration` and `bcst_ReleaseHoldsWorkItem`. The upgrade also drops the `LoadCommandID` column from the work item tables `bc_AdvanceExpiration` and `bc_ReleaseHoldsWorkItem`.

Upgrading NetDistributedToInvoiceItems

Trigger to upgrade the `NetDistributedToInvoiceItems` in `BaseDist`. Prior to 8.0, the `NetDistributedToInvoiceItems` includes both the `AmountDistributed` and all of the `SuspenseAmount`. In 8.0, the `NetDistributedToInvoiceItems` stores only the `AmountDistributed` and a new `NetInSuspense` attribute has been added to store the `SuspenseAmount`. The upgrade updates `NetDistributedToInvoiceItems` and add a new column for `NetInSuspense`.

Deleting AgencyCycleException

The upgrade updates the `AgencyCycleProcess` with exception comments from the `AgencyCycleException` entity. The upgrade then deletes `bc_AgencyCycleException` and `bct1_AgencyCycleException`.

Renaming ModifiedFromSPI to ModifiedFromBSDI

The upgrade renames `bc_ModifiedFromSPI` to `bc_ModifiedFromBSDI` to reflect moving the edge foreign key `ModifiedFrom` from `SuspensePaymentItem` to `BaseSuspDistItem`.

Dropping InvoiceID from BaseDist

The upgrade drops the `bc_BaseDist.InvoiceID` column.

Removing Edge Foreign Key from AgencyCyclePayment to AgencyCyclePromise

The upgrade removes the edge foreign key from `AgencyCyclePayment` to `AgencyCyclePromise` when the payment was reversed and promise unapplied. As of version 8.0, unapplying the promise when reversing the payment removes this link and sets the `AppliedDate` to `null`. Prior to 8.0, unapplying only set `AppliedDate` to `null`. This affected the `isApplied` method, which uses the existence of the link to a payment in 8.0.

Adding the MultiTAccountPattern Subtype

The upgrade updates the database to accommodate the `MultiTAccountPattern` subtype. The upgrade sets the `bc_TAccountPattern` columns `ChargeRollup` and `TAccountLazyLoaded` to nullable. These columns are only required for the `SingleTAccountPattern` subtype. The upgrade then changes the `Subtype` on `TAccountPattern` records to `SingleTAccountPattern`. The upgrade adds a new Designated Unapplied T-account pattern.

Setting Money Received for Reversal Transactions

The upgrade sets the foreign key on a money received context (`bc_AgencyMoneyRcvdContext` or `bc_DBMoneyRcvdContext`) to the appropriate money received for any money received reversal transactions. For `bc_AgencyMoneyRcvdContext`, the upgrade sets the `PaymentMoneyReceivedID` foreign key. For `bc_DBMoneyRcvdContext`, the upgrade sets the `DirectBillMoneyRcvdID` foreign key.

Adding Subtype Column to TAccount

The upgrade adds a `Subtype` column to `bc_TAccount`. In BillingCenter 8.0, the `TAccount` entity has a `MultiTAccount` subtype.

Creating Policy-level Billing Charge Patterns

The upgrade adds the `PolicyLateFee`, `PolicyPaymentReversalFee`, and `PolicyRecapture` charge patterns and associated T-account patterns. These charge patterns are for policy-level billing.

Adding UnappliedFund to Entities

The upgrade adds an `UnappliedFundID` foreign key column to the following entities:

- `AccountDisbursement`
- `AcctNegativeWriteoff`
- `Credit`
- `DirectBillMoneyRcvd`
- `FundsTracker`
- `FundsTransfer`
- `ZDBMRReversal`
- `ZeroDollarDBMR`

The upgrade populates the `UnappliedFund` column with the default unapplied T-account.

Removing TransferTransaction Subtypes and Adding Foreign Keys to TransferTxContext

The upgrade converts all `TransferTransaction` subtypes (`AcctProdTransfer`, `AcctsFundsTransferred`, `ProdAcctTransfer`, `ProducerFundsTransTxn`) to `TransferTransaction` and drops the subtypes. The upgrade also adds `SourceUnappliedTAccountID` and `TargetUnappliedTAccountID` foreign keys to `TransferTxContext`. The upgrade then populates these columns.

Adding ReturnPremiumPlan

The upgrade adds the `ReturnPremiumPlan` subtype of `Plan` and links each `PolicyPeriod` to the `ReturnPremiumPlan`. A return premium plan governs how and when negative policy-level charges, otherwise known as returned premiums, are used to pay other charges in the system.

Converting Zero Dollar Agency Bill Payment Distributions to Credit Distributions

The upgrade converts zero dollar agency bill payment distributions to credit distributions. If the amount of the `MoneyReceived` is zero, then the upgrade sets the `Subtype` to `ZeroDollarAMR`. If the `PaymentInstrument` for the `MoneyReceived` has a `PaymentMethod` that is not `Producer Unapplied`, then the upgrade sets the `PaymentInstrument` for the `MoneyReceived` as follows:

If there are no `PaymentInstruments` with `PaymentMethod` of `Producer Unapplied` associated with the `Producer` of the `MoneyReceived`, the upgrade sets `PaymentInstrument` to a default payment instrument for `Producer Unapplied`.

If a producer is associated with the `PaymentInstruments` with `PaymentMethod` of `Producer Unapplied`, the upgrade sets `PaymentInstrument` to a `PaymentInstrument` associated with the producer. If there are multiple associated `PaymentInstruments`, then the one with the lowest ID is used.

The associated `Transaction` and `LineItems` are deleted in a later step.

Converting Non-zero ZeroDollarAMRs to AgencyBillMoneyRcvds

For `bc_BaseMoneyReceived` records that have Subtype of `ZeroDollarAMR` and an Amount that is not zero, the upgrade sets the Subtype to `AgencyBillMoneyRcvd`.

Converting ZeroDollarReversals to ZeroDollarDMRs

For `bc_BaseMoneyReceived` records that have Subtype of `ZeroDollarReversal`, the upgrade sets the Subtype to `ZeroDollarDMR`.

Converting Non-zero DirectBillMoneyRcvds to ZeroDollarDMRs

For `bc_BaseMoneyReceived` records that have Subtype of `DirectBillMoneyRcvd` and an Amount that is not zero, the upgrade sets the Subtype to `ZeroDollarDMR`.

Deleting Transactions and Line Items for Records with Zero Amount

Deletes `Transactions` and associated `LineItems` for `DirectBillMoneyReceivedTxn` and `AgencyBillMoneyReceivedTxn` records with an amount of zero.

Converting Non-zero ZeroDollarDMRs to DirectBillMoneyRcvds

For `bc_BaseMoneyReceived` records that have Subtype of `ZeroDollarDMR` and an Amount that is not zero, the upgrade sets the Subtype to `DirectBillMoneyRcvd`.

Renaming FundsSourceType and FundsUseType Account Columns

The upgrade renames `bc_FundsSourceType.Account` to `bc_FundsSourceType.UnappliedFund` and `bc_FundsUseType.Account` to `bc_FundsUseType.UnappliedFund`. Both typekeys are used for unapplied fund balance forward funds trackers.

Dropping AccountID from FundsTracker

The upgrade drops the column `bc_FundsTracker.AccountID`.

Moving ReportingGroupID Foreign Key from Account to UnappliedFunds

The upgrade moves the `ReportingGroupID` foreign key from `bc_Account` to `bc_UnappliedFunds`.

Updating General BillingInstruction Records

In 8.0, the supertype of `General BillingInstruction` has changed from `BaseGeneral` to `ExistingPlcyPeriodBI`, so `General` uses different names for the same data now. The upgrade makes the following updates to `bc_BillingInstruction` records with Subtype of `General`:

- Sets the `ModificationDate` to the `BillingInstructionDate`.
- Sets the `AssociatedPolicyPeriodID` to the `PolicyID`.
- Sets the `BillingInstructionDate` to `null`.
- Sets the `PolicyID` to `null`.

Moving PolicyPeriodID to NewPolicyPeriodID for NewPlcyPeriodBI Subtypes

In 8.0, the individual `PolicyPeriodID` fields on `NewPlcyPeriodBI` subtypes were rolled up into a `NewPolicyPeriodID` field. The upgrade moves the data into the new column. This affects the following subtypes:

- Issuance
- NewRenewal
- Renewal
- Rewrite

Adding PCPublicID to Policy

The upgrade adds nullable column `PCPublicID` to `bc_Policy`. This column is used for integration with PolicyCenter.

Dropping DistributedDenorm from BaseMoneyReceived

The upgrade drops the `DistributedDenorm` column from `bc_BaseMoneyReceived`.

Dropping RenewalNumber from PolicyPeriod

The upgrade drops the `RenewalNumber` column from `bc_PolicyPeriod`.

Dropping ReceivableAgingWorkItem

The upgrade checks that there are no work items in the `bc_ReceivableAgingWorkItem` table. If there are no work items, the upgrade drops the `bc_ReceivableAgingWorkItem` table.

Dropping ReceivableAging

The upgrade drops `bc_ReceivableAging`.

Adding Producer Code to AgencyPaymentItem

The upgrade adds the producer code to `AgencyPaymentItem` if it can find a related producer code.

Populating Currency for All Monetary Amounts

The upgrade populates the `Currency` field of all `MonetaryAmount` records in the system. The upgrade sets the value of the column to the default currency.

Renaming Deferred Upgrade Batch Process

The upgrade renames the `DeferredUpgrade` batch process type to `DeferredUpgradeTasks`.

Truncating bc_Dynamic_Assign

The upgrade truncates the `bc_Dynamic_Assign` table.

Dropping bc_t1_Template

The upgrade drops the `bc_t1_Template` table.

Dropping Columns from WorkItem Tables

The upgrade drops the `AvailableSince` and `LastUpdateTime` columns from all `bc_WorkItem` tables.

Upgrading Shared Typekey Data

The upgrade checks for subtypes with typekeys that have the same field name, different column names, and only one column exists in the database. If any such records exist, the upgrade moves the data to the correct column.

Renaming LOBCode Typekey

The upgrade renames the LOBCode typekey HomeOwners to Homeowners.

Dropping DunningInterval from Plan

The upgrade drops the bc_Plan.DunningInterval column.

Adding ListBillAccountExcessTreatment to ReturnPremiumPlan

The upgrade adds a ListBillAccountExcessTreatment column to all ReturnPremiumPlan instances with value of POLICY_PAYER_UNAPPLIED.

Adding PaymentAllocationPlans

The upgrade adds PaymentAllocationPlans representing each distribution limit and links to each Account according to the previous DistributionLimitType of the Account.

Updating Denormalized Fields on InvoiceItem

The upgrade updates denormalized fields CanBePaidMoreByAgencyBill and CanBePromisedMoreByAgencyBill on bc_InvoiceItem to indicate whether or not each invoice item can be paid or promised any more with agency billing.

Associating Users with User-specific Custom Authority Limit Profiles

The upgrade adds records to bc_CustomALPUser to associate users with user-specific custom authority limit profiles and deletes bc_AuthorityLimitProfile.Custom if it exists.

Creating and Updating AccountContext.UnappliedFundID

The upgrade creates and updates the bc_AccountContext.UnappliedFundID foreign key. For AccountContext records, the upgrade sets the UnappliedFundID to the DefaultUnappliedFund of the associated Account. For DisbPaidContext records, the upgrade sets the UnappliedFundID to the UnappliedFund of the associated AccountDisbursement. For AccountNegativeWriteoffContext records, the upgrade sets the UnappliedFundID to the UnappliedFund of the associated AcctNegativeWriteoff.

Adding Payment Allocation Privileges Included with BillingCenter 8.0.1

The upgrade adds the following payment allocation privileges:

Privilege	Roles
payallocplancreate	general_admin
	plan_admin
	superuser

Privilege	Roles
payallocplanedit	general_admin plan_admin superuser
payallocplanview	billing_clerical billing_manager commissions_admin finance_manager general_admin plan_admin superuser underwriter

Adding Direct Collector Role

The upgrade adds the Direct Collector role.

Dropping PolicyDlnqProcessID from LegacyDlnqWorkItem

The upgrade drops the `bc_LegacyDlnqWorkItem.PolicyDlnqProcessID` column.

Viewing Detailed Database Upgrade Information

BillingCenter includes an [Upgrade Info](#) page that provides detailed information about the database upgrade. The [Upgrade Info](#) page includes information on the following:

- version numbers before and after the database upgrade
- configuration parameters used during the database upgrade
- SQL queries for version checks that test if the database is in condition to be upgraded
- changes made to specific tables, including which version triggers modified the table or its data and the SQL statement executed to make each change
- version triggers that the upgrade ran, including which tables the trigger ran against, a description, the SQL statement run against each table and the start and end time
- a list of upgrade steps, including the table on which the step operated
- a table registry including table IDs before and after upgrade

The database upgrade deletes upgrade instrumentation information for prior database upgrades. If the database upgrade detects any prior upgrade instrumentation data, it reports a warning and deletes the data. If you have run previous database upgrades, and you want to preserve upgrade instrumentation details, download this information.

To download upgrade instrumentation details

1. Start the BillingCenter server if it is not already running.
2. Log in to BillingCenter with the superuser account.
3. Press ALT+SHIFT+T to access **System Tools**.
4. Click **Info Pages**.
5. Select **Upgrade Info** from the **Info Pages** drop-down.

6. Click **Download** to download a ZIP file containing the detailed upgrade information.

Dropping Unused Columns on Oracle

By default, the BillingCenter database upgrade on Oracle marks columns that have been removed from the data model as unused. Marking a column unused is a faster operation than dropping a column. Because these columns are not physically dropped from the database, the space used by these columns is not released immediately to the table and index segments.

You can configure the upgrade to drop removed columns immediately by setting the `deferDropColumns` parameter to `false` before running the database upgrade. This parameter is within the `<upgrade>` block of the `<database>` block of `database-config.xml`.

If you did not set `deferDropColumns` to `true` before the upgrade, perform the procedure in this topic to drop unused columns after the upgrade. You can drop the unused columns after the upgrade during off-peak hours to free the space. BillingCenter does not have to be shutdown to perform this maintenance task. You can drop all unused columns in one procedure, or you can drop unused columns for individual tables.

To drop all unused columns

1. Create the following Oracle procedure to purge all unused columns:

```
DECLARE
    dropstr VARCHAR2(100);
    CURSOR unusedcol IS
        SELECT table_name
        FROM user_unused_col_tabs;
BEGIN
    FOR tabs IN unusedcol LOOP
        dropstr := 'alter table '
            || tabs.table_name
            || ' drop unused columns';
        EXECUTE IMMEDIATE dropstr;
    END LOOP;
END;
```

2. Run the procedure during a period of relatively low activity.

To drop unused columns for a single table (or all tables)

1. Start the server to run the schema verifier. The schema verifier runs each time the server starts. If there are unused columns, the schema verifier reports a difference between the physical database and the data model. The schema verifier reports the name of each table and provides an SQL command to remove unused columns from each table.

2. Run the SQL command provided by the schema verifier. This command has the following format:

```
ALTER TABLE tableName DROP UNUSED COLUMNS
```

Exporting Administration Data for Testing

Guidewire recommends that you create a small set of administration data from an upgraded data set. Use this data for development and testing of rules and libraries with BillingCenter 8.0.4. This procedure is optional.

You might have already created an upgraded administration data set by following the procedure “Upgrading Administration Data for Testing” on page 250. If you followed that procedure, or you do not want an administration-only data set for testing purposes, you can skip this topic.

To create an administration data set for testing

1. Export administration data from your upgraded production database.
 - a. Start the BillingCenter 8.0.4 server by navigating to `BillingCenter/bin` and running the following command:
`gwbc dev-start`
 - b. Open a browser to BillingCenter 8.0.4.
 - c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
 - d. Click the **Administration** tab.
 - e. Click → Utilities → Export Data.
 - f. Select the **Admin** data set to export.
 - g. Click **Export** to download the `admin.xml` file.
2. Create a new database account for the development environment on a database management system supported by BillingCenter 8.0.4. See the *Guidewire Platform Support Matrix* for current system and patch level requirements. The *Guidewire Platform Support Matrix* is available from the Guidewire Resource Portal at <http://guidewire.custhelp.com>.
See “Configuring the Database” on page 23 in the *Installation Guide* for instructions to configure the database account.
3. Install a new BillingCenter 8.0.4 development environment. Connect this development environment to the new database account that you created in step 2. See the *BillingCenter Installation Guide* for instructions.
4. Copy the `admin.xml` file that you exported to a location accessible from the new development environment.
5. Create an empty version of `importfiles.txt` in the `modules/configuration/config/import/gen` directory of the new development environment.
6. Create empty versions of the following CSV files:
 - `activity-patterns.csv`
 - `authority-limits.csv`
 - `reportgroups.csv`
 - `roleprivileges.csv`
 - `rolereportprivileges.csv`Leave `roles.csv` as the original complete file.
7. Import the administration data into the new database:
 - a. Start the BillingCenter 8.0.4 development server by navigating to `BillingCenter/bin` and running the following command:
`gwbc dev-start`
 - b. Open a browser to BillingCenter 8.0.4.
 - c. Log on as a user with the `viewadmin` and `soapadmin` permissions.
 - d. Click the **Administration** tab.
 - e. Click → Utilities → Import Data.
 - f. Click **Browse....**
 - g. Select the `admin.xml` file that you exported from the upgraded production database and modified.
 - h. Click **Open**.

Upgrading Phone Numbers

BillingCenter 8.0 has a different format for phone numbers. Each phone number type has two additional fields in 8.0: a country code and an extension. The country code is a typekey to the `PhoneCountryCode` typelist, which is a list of regions and their regional phone codes.

BillingCenter 8.0 provides a plugin interface, `IPhoneNormalizerPlugin`, that you can use to customize the conversion of legacy phone numbers to the 8.0 standard. The interface includes the following method signatures:

- `isPossibleNumber(String) : boolean`
- `isPossibleNumberWithExtension(String) : boolean`
- `normalizeNumberIfPossible(String) : String`
- `parsePhoneNumber(String) : GWPhoneNumber`
- `formatPhoneNumber(GWPhoneNumber number) : String`
- `normalizePhoneNumbersInBean(KeyableBean) : void`
- `normalizePhoneNumbersInArchive(IArchivedEntity, java.util.List<PhoneColumnProperties>) : void`

The plugin is called by the Phone Number Normalizer work queue. The plugin is also called whenever an entity containing a phone number is modified in BillingCenter.

Guidewire provides a default implementation of the plugin, `gw.api.phone.DefaultPhoneNormalizerPlugin`. If you disabled the phone number input mask or imported phone numbers, you might need to customize the plugin implementation. If you added new phone fields on existing objects or extension objects, customize or extend the plugin implementation to handle the additional phone fields. The `gw.api.util.PhoneUtil` class contains helper methods to facilitate formatting and parsing phone number records.

Define both `isPossibleNumber` and `isPossibleNumberWithExtension` methods to be very loose, non-country specific, validations. These methods essentially just need to check if the passed string could be a number in any country.

In the default phone normalizer plugin implementation, the `normalizeNumberIfPossible` method strips all decorator and formatting characters from the number if `isPossibleNumber` returns `true`. The normalizer ignores all numeric characters as well as + and * characters.

The `parsePhoneNumber` and `formatPhoneNumber` methods are used to convert between BillingCenter 7.0 and BillingCenter 8.0 phone numbers. The `parsePhoneNumber` method parses a string into a `GWPhoneNumber` object if possible. `GWPhoneNumber` is an interface that defines a standard BillingCenter 8.0 phone number object. See the Javadoc for further details. The `parsePhoneNumber` method is for converting phone numbers from versions prior to 8.0 to the 8.0 standard. The `formatPhoneNumber` method formats a `GWPhoneNumber` object into a single string. The `formatPhoneNumber` method is for converting 8.0 phone numbers to the 7.0 standard.

The plugin only normalizes a phone number if `isPossibleNumber` returns `true`. If `isPossibleNumber` returns `true`, the plugin calls `parsePhoneNumber` to convert the number to a `GWPhoneNumber` object.

By default, the maximum length of a phone number extension field is four. You can change the maximum length of phone number extensions by specifying an `extensionLength` parameter on the plugin implementation.

1. In Studio, open configuration → config → Plugins → registry → `IPhoneNormalizerPlugin.gwp`.
2. Click the Add Parameter  icon next to Parameters.
3. Enter `extensionLength` for the key.
4. Enter a numeric value for value.

You can call the phone normalizer plugin when adding a contact record from an external system to convert the phone number to the BillingCenter 8.0 standard. You might need to customize the plugin depending on the format of your source data.

The Phone Number Normalizer work queue generates work items for phone numbers with a country code of `unparseable` or `null`, indicating that the plugin has not yet processed the number.

If you are using ContactManager, run the Phone Number Normalizer work queue for ContactManager first. Then run the Phone Number Normalizer work queue for BillingCenter. Phone numbers in BillingCenter may become out of sync with ContactManager while the ContactManager Phone Number Normalizer work queue is running. It is safe to sync contacts in BillingCenter that become out of sync with ContactManager. When you run the Phone Number Normalizer work queue for BillingCenter, it skips the previously synced records.

Eventually, run the Phone Number Normalizer work queue for all of your Guidewire applications.

For performance reasons, run the Phone Number Normalizer work queue at off-peak hours. Some functionality, such as ContactManager's de-duplication feature could perform poorly while the Phone Number Normalizer work queue runs. You could see Concurrent Data Change Exceptions if you modify an existing contact at the same time as the Phone Number Normalizer work queue. If this occurs, reload the contact and attempt the update again.

Final Steps After The Database Upgrade is Complete

This section describes procedures to run after you have completed the upgrade procedure and migration of configurations and integrations. The processes and checks in this section provide you with a benchmark of the upgraded system. Completing these steps is particularly important to going live in a production environment.

Use these procedures to revalidate the database:

- “Validating the Database Schema” on page 253
- “Checking Database Consistency” on page 254 including “Checking that Contacts Have Unique Addresses” on page 318
- “Creating a Data Distribution Report” on page 254
- “Generating Database Statistics” on page 255. You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded BillingCenter in a production environment.
- “Reenabling Database Logging” on page 319
- “Backing up the Database After Upgrade” on page 319

Checking that Contacts Have Unique Addresses

An `Address` cannot be shared by more than one `Contact`. BillingCenter 8.0 includes a commit-time check that does not allow a shared reference to an address instance even when one of the referring `Contact` or `ContactAddress` instances is retired. If you have multiple contacts at the same address, you can create separate address instances with the same field values.

A database consistency check on the `Contact` entity reports an error if it detects multiple `Contact` records using the same `PrimaryAddress`.

Before using BillingCenter 8.0.4 in production, run database consistency checks to find any instances of shared references to address instances. If the consistency check reports shared addresses, contact Guidewire Support for assistance fixing your database.

Completing Deferred Upgrade

If you have archiving enabled, and you did not set `deferCreateArchiveIndexes` to `false`, run the `Deferred Upgrade Tasks` batch process as soon as possible after the completion of the upgrade. To run the `Deferred Upgrade Tasks` batch process, use the `admin/bin/maintenance_tools` command:

```
maintenance_tools -password password -startprocess deferredupgradetasks
```

Reenabling Database Logging

You might have disabled logging of direct insert and create index operations during the database upgrade. After you complete the database upgrade successfully, you can reenable logging by setting `allowUnloggedOperations` to `false` in the `<upgrade>` block. For example:

```
<database ...>
...
<upgrade allowUnloggedOperations="false">
...
</upgrade>
</database>
```

For SQL Server, if you changed the recovery model from Full to Simple or Bulk logged during the upgrade, you can revert the recovery model. If you deferred migrating to 64-bit IDs, you might disable logging again when you perform the migration.

Migrating to 64-bit IDs After Upgrade (SQL Server Only)

If you did not change the `MigrateToLargeIDsAndDatetime2` parameter to `true` before the upgrade, complete the migration to 64-bit primary key and foreign key identifiers after the upgrade. This ensures that there are enough unique identifiers available for BillingCenter.

To migrate primary key and foreign key identifiers and date columns after the upgrade

1. Open Studio for BillingCenter 8.0.4.
2. Open `configuration` → `config` → `config.xml`.
3. Change the value of the `MigrateToLargeIDsAndDatetime2` parameter to `true`.
4. Save your changes.
5. Open `configuration` → `config` → `database-config.xml`.
6. Add the `allowUnloggedOperations` attribute to the `upgrade` element if you have not already added it.

```
<upgrade allowUnloggedOperations="true" ... />
```

If you do not require full logging, set `allowUnloggedOperations` to `true` to improve performance of the conversion. Set the SQL Server recovery model to Simple or Bulk logged during the migration.

If you do require full logging due to the presence of solutions such as Database Mirroring, continue to use the Full recovery model and set `allowUnloggedOperations` to `false`.

See “Disabling SQL Server Logging” on page 274.

7. Save your changes.
8. Start the BillingCenter server. BillingCenter performs the migration to 64-bit BIGINT identifiers and `datetime2` date columns.
9. Revert the SQL Server recovery model if you changed it in step 6.

Backing up the Database After Upgrade

Finally, before going live, back up the upgraded database. This provides you with a snapshot of the initial upgraded data set, if an unanticipated event occurs just after going live.

Upgrading Integrations and Gosu from 3.0.x

This topic lists the tasks to upgrade to this release. The tasks are presented in tables, according to when you perform the tasks. You can print these tables to use them as checklists during the upgrade.

This topic includes:

- “Overview of Upgrading Integration Plugins and Code” on page 321
- “Tasks Required Before Starting the Server” on page 322
- “Tasks Required Before Deploying a Production Server” on page 324
- “Tasks Required Before the Next Upgrade” on page 324

Overview of Upgrading Integration Plugins and Code

This topic provides a high level approach to upgrading integration plugins and code. Review this topic, then proceed to the following topics for specific upgrade steps:

- Tasks Required Before Starting the Server
- Tasks Required Before Deploying a Production Server
- Tasks Required Before the Next Upgrade

As part of integration, developers add third-party libraries (JAR files) to the Java API and SOAP API libraries to compile their code. During the upgrade phase, segregate these third-party libraries from the Java and SOAP libraries. Initially, it is more practical to use these third-party libraries as is during the upgrade process. Later, you can upgrade Java API and SOAP API libraries separately, along with any ramification to the code, as necessary.

The database upgrade usually matures over the initial cycles of the upgrade process. If the integration code upgrade starts at the same time, regenerating the SOAP API and Java API might not yield the final versions of these libraries. Consult with the database upgrade team to determine when to regenerate the SOAP API and Java API for more current libraries.

Integration upgrade steps

1. Create a project with the code at hand after segregating third-party libraries from the Java and SOAP libraries.
2. Ensure you can successfully compile.
3. Create a backup copy of the project.
4. Replace the default Java and SOAP libraries with upgraded libraries. Leave third-party libraries as is.
5. Update to the correct version of Java. See “Installing Java” on page 40 in the *Installation Guide*.
6. Many classes will fail to compile correctly. The error list is literally the technical upgrade. It needs to be sorted and addressed.

The most commonly encountered compiler failures during upgrade are described in the following table:

Issue	Example	How to approach upgrade
Java upgrades	Refer to Java documentation.	
Changes in object construction	<code>EntityFactory.getEntityFactory().newEntity() → EntityFactory.getInstance().newEntity()</code>	Identify the changes and then use a utility to find and replace throughout the code base.
Name changes	<code>gscript → gosu</code>	Identify the changes and then use a utility to find and replace throughout the code base.
Discontinued support of utilities available in previous versions	<code>com.guidewire.util.FileSystem</code> and <code>com.guidewire.util.FileUtil</code>	Carry the old implementation forward as a third-party library.
Class relocation	<code>com.guidewire.logging.SystemOutLogger → gw.util.SystemOutLogger</code>	Locate the new package using searches or a utility such as <code>scanzip</code> in the new <code>soap-api</code> and <code>java-api</code> directories.
Additional interface methods	The <code>IDocumentContentSource</code> interface gained additional methods: <ul style="list-style-type: none">• <code>getDocumentContentsInfoForExternalUse()</code>• <code>isInboundAvailable()</code>• <code>isOutboundAvailable()</code>	Review the <i>New and Changed Guide</i> for additional methods on key interfaces used in your integration plugins.
Functional changes (most involved to upgrade)		Understand the changes and what the code is trying to do and modify your code accordingly. Review the <i>New and Changed Guide</i>

Tasks Required Before Starting the Server

The following table contains things you must do before you start the server.

Tasks	For more information...
<input type="checkbox"/> Follow the basic upgrade procedure.	“Upgrading the BillingCenter 3.0.x Configuration” on page 203
<input type="checkbox"/> Change references to SOAP and WSDL packages to paths that include version numbers.	“SOAP Implementation Classes and WSDL Packages Include Version” on page 127 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Change properties that use different types for getters and setters to use the same types.	“Mismatched property Getter/Setter Types” on page 107 in the <i>New and Changed Guide</i>

<input checked="" type="checkbox"/> Tasks	For more information...
<input type="checkbox"/> Change overridden generic functions that have a different parameterization to match the parameterization of the overridden method declaration.	“Overriding a Generic Function with a non-Generic function” on page 107 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Change overrides of getters that use different types than the superclass to use the same types.	“Covariantly Overriding the Getter Half of a Writable Property” on page 107 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Change enhancement methods that override methods defined on superclasses of the enhancements.	“Overriding an Enhancement Method” on page 108 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Change overloaded methods that vary by non-Java-backed types as arguments.	“Method Overloading Involving Non-Java-backed Types as the Arguments” on page 108 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Add type declarations to variables that you initialize to no value.	“Variables With No Type Cannot Initialize to Null” on page 109 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Remove explicit methods that collide with implicit getter and setter methods for Gosu properties.	“Properties Must Not Conflict with Explicit Getter or Setter Methods” on page 109 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Remove void functions from expressions and put them in stand alone statements.	“Do Not Use the Return Value of a Void Function In an Expression” on page 110 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Change imports of types that use relative paths to use fully qualified paths.	“Relative Imports Discouraged, and Now Sometimes Require Fully-Qualified Type Names” on page 111 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code for package-local Java classes and how behavior might change.	“Accessing Package-local Java Classes from Gosu classes in the Same Package” on page 112 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code for changes to null safety of arithmetic operators.	“Standard Arithmetic Operators Are No Longer Null-safe” on page 112 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code for changes to array casts.	“Array Casts” on page 113 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code for changes to primitive property short-circuiting	“Primitive Property Short-Circuiting” on page 113 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code for changes to class loading.	“Class Loading and Initialization Ordering” on page 114 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code for changes to arrays of non-Java-backed types with <code>typeof</code> or <code>TypeSystem.getFromObject(o)</code> .	“Arrays of Non-Java-backed Types with ‘typeof’ or ‘TypeSystem.getFromObject(o)’” on page 114 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code for changes to Java code that catches exceptions from Gosu.	“Catching Exceptions in Java When Gosu throws Exceptions” on page 115 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code for changes to private variables on superclasses with the same name as variables on subclasses.	“Private Variables on Superclasses with the Same Name as a Variable on the Subclass” on page 116 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code to remove any Gosu interceptors.	“Interceptors Removed” on page 117 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Change your code that calls or implements the <code>onChargesPaid</code> method of the <code>IDelinquencyProcessExtensions</code> plugin interface to take a list of <code>ChargePaidFromUnapplied</code> instances instead of <code>AbstractChargePaidTxn</code> instances.	“Changes to Delinquency Processing Customization” on page 130 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Refactor any code that customized the behavior of the <code>IBillingInstruction</code> plugin interface and change your implementation accordingly to use the new charge slicers.	“Changes to Billing Instruction Customization” on page 130 in the <i>New and Changed Guide</i>

 Tasks	For more information...
<input type="checkbox"/> Refactor any code that made payments through the IBillingCenterAPI web service and change your implementation accordingly.	"Changes to Making Payments" on page 131 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Refactor any code that might be affected by changes to the IIInvoice and IIInvoiceStream plugins and change your implementation accordingly.	"Changes to the Invoice and Related Plugins" on page 132 in the <i>New and Changed Guide</i>

Tasks Required Before Deploying a Production Server

The following table contains tasks to complete before starting the server and changes to familiarize yourself with before deploying a server to a production environment.

 Tasks	For more information...
<input type="checkbox"/> Review changes to the IPolicyPeriod and IPolicySystem plugin interfaces and change your implementation accordingly.	"Changes to the Policy Period and Policy System Plugins" on page 133 in the <i>New and Changed Guide</i>

Tasks Required Before the Next Upgrade

The following table contains tasks required before the next upgrade. For example, if you used APIs that are now deprecated, begin rewriting your code to avoid use of deprecated APIs. Guidewire will remove these APIs in a future release.

 Tasks	For more information...
<input type="checkbox"/> Update your plugin implementation classes to the version of your Guidewire application.	"Guidewire InsuranceSuite Plugin Implementations are Versioned" on page 127 in the <i>New and Changed Guide</i>
<input type="checkbox"/> In try/catch statements, rewrite undeclared exception types to catch type Exception or a more specific subtype. Review code in the catch block to ensure it does not unwrap the exception.	"Checked Exceptions Changes in Gosu" on page 110 in the <i>New and Changed Guide</i>
