

# Guidewire PolicyCenter®

## PolicyCenter Upgrade Guide

RELEASE 8.0.3



Copyright © 2001-2014 Guidewire Software, Inc. All rights reserved.

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

All other trademarks are the property of their respective owners.

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

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

Product Name: Guidewire PolicyCenter

Product Release: 8.0.3

Document Name: PolicyCenter Upgrade Guide

Document Revision: 18-November-2014

# Contents

<b>About PolicyCenter Documentation .....</b>	<b>13</b>
Conventions in This Document .....	14
Support .....	14

## Part I

### Planning the Upgrade

<b>1 Planning Your PolicyCenter Upgrade.....</b>	<b>17</b>
Supported Starting Version.....	18
Upgrading from Version 3.0.....	18
Upgrading Language Packs .....	19
Roadmap for Planning the Upgrade .....	19
Upgrade Assessment.....	19
Preparing for the Upgrade.....	21
Project Inception.....	22
Design and Development .....	23
System Test.....	23
Deployment and Support .....	23

## Part II

### Upgrading from 8.0.x

<b>2 Upgrading the PolicyCenter 8.0.x Configuration.....</b>	<b>27</b>
Overview of ContactManager Upgrade .....	28
Obtaining Configurations and Tools.....	28
Viewing Differences Between Base and Target Releases .....	29
Specifying Configuration and Tool Locations .....	29
Creating a Configuration Backup .....	32
Backing up the Configuration.....	32
Backing up the Product Model .....	32
Removing Patches.....	32
Removing Language Packs.....	32
Updating Infrastructure.....	32
Launching the PolicyCenter 8.0.3 Configuration Upgrade Tool .....	33
Restarting the Configuration Upgrade Tool .....	33
Configuration Upgrade Tool Automated Steps .....	34
Copying Custom Rules and Adding PolicyCenter 8.0.3 Default Rules .....	34
Renaming Parameter.....	34
Updating Product Model Structure .....	34
Using the PolicyCenter 8.0.3 Upgrade Tool Interface .....	35
Filters .....	36
Configuration File Tree.....	40
File Details Panel .....	40
Accepting Files that Do Not Require Merging .....	41
Merging and Accepting Files .....	41
Merging Product Model Files .....	41
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
Updating Product Model API Calls .....	45
Merging Display Properties .....	46
Upgrading Rules to PolicyCenter 8.0.3 .....	47
Translating New Display Properties and Typecodes .....	48
Validating the PolicyCenter 8.0.3 Configuration .....	48
Using Studio to Verify Files .....	48
Starting PolicyCenter and Resolving Errors.....	49
Importing Policy Forms .....	49
Building and Deploying PolicyCenter 8.0.3.....	50
<b>3 Upgrading the PolicyCenter 8.0.x Database.....</b>	<b>51</b>
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
Customizing the Upgrade .....	60
Running Custom Version Checks and Triggers .....	61
IDatamodelUpgrade API Examples .....	64
Upgrading Archived Entities.....	71
Disabling the Scheduler .....	73
Suspending Message Destinations .....	74
Configuring the Database Upgrade.....	74
Adjusting Commit Size for Encryption .....	74
Configuring Version Trigger Elements.....	75
Deferring Creation of Nonessential Indexes.....	76
Configuring the Upgrade on Oracle .....	77
Configuring the Upgrade on SQL Server .....	79
Downloading Database Upgrade Instrumentation Details .....	80
Checking the Database Before Upgrade.....	81

Starting the Server to Begin Automatic Database Upgrade .....	81
Test the Database Upgrade .....	82
Integrations and Starting the Server .....	83
Understanding the Automatic Database Upgrade .....	83
Version Trigger Descriptions .....	84
Viewing Detailed Database Upgrade Information .....	85
Dropping Unused Columns on Oracle .....	86
Reloading Rating Sample Data .....	87
Exporting Administration Data for Testing .....	87
Final Steps After The Database Upgrade is Complete .....	89
Checking that Contacts Have Unique Addresses .....	89
Completing Deferred Upgrade .....	89
Backing up the Database After Upgrade .....	89
<b>4 Upgrading PolicyCenter from 8.0.x for ContactManager .....</b>	<b>91</b>
Manually Upgrading PolicyCenter to Integrate with ContactManager .....	91
File Changes in PolicyCenter Related to ContactManager .....	92
Web Service Version Changes .....	92
PolicyCenter Can Generate and Send Unique IDs for New Contacts .....	92
<b>5 Upgrading ContactManager from 8.0.x .....</b>	<b>93</b>
Manually Upgrading the ContactManager Configuration .....	93
Manually Configuring Changed Files .....	94
BillingCenter Web Services Version Change .....	94

## Part III Upgrading from 7.0.x

<b>6 Upgrading the PolicyCenter 7.0.x Configuration .....</b>	<b>99</b>
Overview of ContactManager Upgrade .....	100
Obtaining Configurations and Tools .....	100
Viewing Differences Between Base and Target Releases .....	101
Specifying Configuration and Tool Locations .....	102
Creating a Configuration Backup .....	104
Backing up the Configuration .....	104
Backing up the Product Model .....	104
Removing Patches .....	104
Removing Language Packs .....	104
Updating Infrastructure .....	104
Launching the PolicyCenter 8.0.3 Configuration Upgrade Tool .....	105
Restarting the Configuration Upgrade Tool .....	105

Configuration Upgrade Tool Automated Steps .....	106
Removing Template Pages .....	106
Updating PCF Files.....	106
Upgrading Work Queue Configuration.....	108
Upgrading Database Configuration.....	108
Splitting Localization.xml into Separate Files for each Locale .....	111
Splitting address-config.xml into Separate Files for each Country .....	111
Splitting zone-config.xml into Separate Files for each Country.....	111
Splitting currencies.xml into Separate Files for each Currency .....	111
Moving Country-based Field Validator Definition Files .....	111
Moving Rules Files up One Directory .....	111
Reformatting Rules for Display in Studio Rules Editor .....	111
Copying Custom Rules and Adding PolicyCenter 8.0.3 Default Rules .....	111
Renaming SOAP Web Services from XML to RWS .....	112
Renaming Plugins from XML to GWP.....	112
Renaming Display Names Files from XML to EN.....	112
Upgrading Display Keys.....	112
Adding nullok="true" to Entity and Extension Foreign Key Columns.....	112
Removing deletefk Attribute from Entity and Extension Foreign Keys.....	112
Setting XML Namespace on Metadata Files .....	112
Upgrading Document Assistant Parameters .....	113
Separating Entities and Typelists .....	113
Adding Default Currency on CovTermOpt and CovTermPack Nodes .....	113
Adding Currency Filters to Choice Lookup Table Configurations.....	113
Adding CovTermLimits to DirectCovTermPattern .....	113
Adding CovTermDefault to OptionCovTermPattern .....	113
Adding Default Currency to PolicyLinePattern .....	113
Setting Default Answer for Questions with BooleanCheckbox Format.....	114
Setting questionPostOnChange to auto.....	114
Normalizing Dates in the Product Model to a Standard Format.....	114
Removing splitOnAnniversary from Product Line Configuration .....	114
Using the PolicyCenter 8.0.3 Upgrade Tool Interface .....	114
Filters .....	115
Configuration File Tree.....	119
File Details Panel .....	119
Accepting Files that Do Not Require Merging .....	120
Merging and Accepting Files .....	120
Merging Product Model Files.....	120
Configuration Merging Guidelines.....	121
Data Model Merging Guidelines .....	122
Updating Data Types for Case Sensitivity .....	122
Merging Typelists – Overview .....	122
Merging Typelists – Simple Typelists .....	123
Merging Typelists – Complex Typelists.....	123
Reviewing Shared Typekey Configuration.....	123
Adding State Typelist Extensions to Jurisdiction .....	124
Merging Entity Extensions .....	124
Reviewing Custom Extensions .....	125
Reconciling the Database with Custom Extensions .....	125
Removing Obsolete Attributes .....	125
Updating Extractable Edge Foreign Keys.....	125
Converting Money to MonetaryAmount.....	125
Updating Product Model API Calls .....	126

Merging PolicyCenter Typelists .....	127
GLCoverageFormType .....	128
PercentDuplicated .....	129
ReceptacleType .....	129
GLStateCostType .....	129
Changes to the Logging API .....	129
Conceptual Changes to Logging .....	129
Instantiating Loggers .....	131
Logging Messages .....	131
Passing Loggers as Parameters .....	131
Merging Enhancements .....	132
Updating PolicyPeriodPlugin.gs .....	133
Consider Enabling Check for Small Cost Changes .....	133
Merging Claim Details PCF Files .....	134
Adding DDL Configuration Options to database-config.xml .....	134
Merging Changes to Field Validators .....	134
Renaming PCF files According to Their Modes .....	135
Merging Display Properties .....	135
Merging Other Files .....	135
Fixing Gosu Issues .....	136
Gosu Case Sensitivity .....	136
Inequality Operator .....	136
Ambiguous Method Calls .....	137
Nested Comments .....	137
Upgrading Rules to PolicyCenter 8.0.3 .....	138
Rules Required for Free Text Search .....	139
Translating New Display Properties and Typecodes .....	140
Validating the PolicyCenter 8.0.3 Configuration .....	140
Using Studio to Verify Files .....	140
Starting PolicyCenter and Resolving Errors .....	140
Importing Policy Forms .....	141
Building and Deploying PolicyCenter 8.0.3 .....	141
<b>7 Upgrading the PolicyCenter 7.0.x Database.....</b>	<b>143</b>
Upgrading Administration Data for Testing .....	144
Identifying Data Model Issues .....	145
Verifying Batch Process and Work Queue Completion .....	146
Purging Data Prior to Upgrade .....	146
Purging Old Messages from the Database .....	146
Purging Completed Workflows and Workflow Logs .....	147
Validating the Database Schema .....	147
Checking Database Consistency .....	148
Creating a Data Distribution Report .....	148
Generating Database Statistics .....	149
Creating a Database Backup .....	150
Updating Database Infrastructure .....	150
Preparing the Database for Upgrade .....	150
Ensuring Adequate Free Space .....	150
Disabling Replication .....	150
Assigning Default Tablespace (Oracle only) .....	150
Setting Linguistic Search Collation .....	151
Deleting CoverageSymbolGroup from Coverage .....	152

Customizing the Upgrade .....	153
Running Custom Version Checks and Triggers .....	153
IDatamodelUpgrade API Examples .....	157
Upgrading Archived Entities .....	163
Disabling the Scheduler .....	165
Suspending Message Destinations .....	166
Configuring the Database Upgrade .....	166
Adjusting Commit Size for Encryption .....	166
Configuring Version Trigger Elements .....	167
Deferring Creation of Nonessential Indexes .....	168
Configuring the Upgrade on Oracle .....	169
Configuring the Upgrade on SQL Server .....	171
Downloading Database Upgrade Instrumentation Details .....	172
Checking the Database Before Upgrade .....	173
Specifying ValueType on Coverage Terms .....	173
Dropping Custom Rating Worksheet Tables .....	174
Starting the Server to Begin Automatic Database Upgrade .....	175
Test the Database Upgrade .....	175
Integrations and Starting the Server .....	176
Understanding the Automatic Database Upgrade .....	176
Version Trigger Descriptions .....	177
Viewing Detailed Database Upgrade Information .....	189
Dropping Unused Columns on Oracle .....	189
Reloading Rating Sample Data .....	190
Exporting Administration Data for Testing .....	191
Upgrading Phone Numbers .....	192
Final Steps After The Database Upgrade is Complete .....	193
Checking that Contacts Have Unique Addresses .....	194
Completing Deferred Upgrade .....	194
Backing up the Database After Upgrade .....	194
<b>8 Upgrading PolicyCenter from 7.0.x for ContactManager .....</b>	<b>195</b>
Configuration File Changes in PolicyCenter .....	196
Manually Upgrading PolicyCenter to Integrate with ContactManager .....	197
Mapping Your Contact Extensions .....	197
Parameter transactionId Removed from ContactManager Web Services .....	198
<b>9 Upgrading ContactManager from 7.0.x .....</b>	<b>199</b>
Database Upgrade Steps in ContactManager .....	199
Preserving MatchSetKey Column Data .....	199
Ensuring that LinkID Is Unique .....	200
Configuration File Changes in ContactManager .....	200
Manually Configuring Changed Files .....	201
<b>Part IV</b>	
<b>Upgrading from 4.0.x</b>	
<b>10 Upgrading the PolicyCenter 4.0.x Configuration .....</b>	<b>209</b>
Obtaining Configurations .....	210
Viewing Differences Between Base and Target Releases .....	211
Specifying Configuration Locations for PolicyCenter 7.0 Upgrade Tool .....	211
Creating a Configuration Backup .....	214
Backing up the Configuration .....	214
Backing up the Product Model .....	214
Removing Patches .....	214

Removing Language Packs.....	214
Updating Infrastructure.....	214
Upgrading the PolicyCenter 4.0 Configuration to 7.0 .....	215
Launching the PolicyCenter 7.0 Configuration Upgrade Tool.....	215
Restarting the Configuration Upgrade Tool .....	215
PolicyCenter 7.0 Upgrade Tool Automated Steps .....	215
Moving Typelist Localizations into typelist.properties Files .....	215
Removing Redundant TTX Files .....	216
Removing searchTypeVisible Attribute from DateCriterionChoiceInputNode .....	216
Copying Display Properties Files into Target Configuration .....	216
Copying Custom Rules and Adding PolicyCenter 8.0.3 Default Rules .....	216
Referencing XSD Files .....	216
Removing AdminTable Delegate from Custom Extensions.....	216
Converting sessiontimeoutsecs Security Element to Parameter .....	216
Removing Redundant Batch Server Parameter.....	217
Upgrading Question Sets and Questions.....	217
Converting Form Pattern XML into Import XML .....	217
Mapping Custom Inference Classes to Form Codes.....	217
Deleting Form Pattern Files and Directories.....	217
Deleting Form Display Keys.....	217
Creating Jurisdiction Typelist .....	217
Renaming System Tables .....	218
Configuring the PolicyCenter 8.0 Upgrade Tool .....	218
Launching the PolicyCenter 8.0 Configuration Upgrade Tool.....	220
Restarting the Configuration Upgrade Tool .....	220

PolicyCenter 8.0.3 Configuration Upgrade Tool Automated Steps .....	220
Removing Template Pages .....	220
Updating PCF Files.....	220
Upgrading Work Queue Configuration.....	223
Upgrading Database Configuration.....	223
Splitting Localization.xml into Separate Files for each Locale .....	225
Splitting address-config.xml into Separate Files for each Country .....	225
Splitting zone-config.xml into Separate Files for each Country.....	225
Splitting currencies.xml into Separate Files for each Currency .....	225
Moving Country-based Field Validator Definition Files .....	226
Moving Rules Files up One Directory .....	226
Reformatting Rules for Display in Studio Rules Editor .....	226
Copying Custom Rules and Adding PolicyCenter 8.0.3 Default Rules .....	226
Renaming SOAP Web Services from XML to RWS .....	226
Renaming Plugins from XML to GWP.....	226
Renaming Display Names Files from XML to EN.....	226
Upgrading Display Keys.....	226
Adding nullok="true" to Entity and Extension Foreign Key Columns.....	227
Removing deletefk Attribute from Entity and Extension Foreign Keys.....	227
Setting XML Namespace on Metadata Files .....	227
Upgrading Document Assistant Parameters .....	227
Separating Entities and Typelists .....	227
Adding Default Currency on CovTermOpt and CovTermPack Nodes .....	228
Adding Currency Filters to Choice Lookup Table Configurations.....	228
Adding CovTermLimits to DirectCovTermPattern .....	228
Adding CovTermDefault to OptionCovTermPattern .....	228
Adding Default Currency to PolicyLinePattern .....	228
Setting Default Answer for Questions with BooleanCheckbox Format.....	228
Setting questionPostOnChange to auto.....	228
Normalizing Dates in the Product Model to a Standard Format.....	228
Removing splitOnAnniversary from Product Line Configuration .....	228
Using the PolicyCenter 8.0.3 Upgrade Tool Interface .....	229
Filters .....	230
Configuration File Tree.....	234
File Details Panel .....	234
Accepting Files that Do Not Require Merging .....	235
Merging and Accepting Files .....	235
Merging Product Model Files.....	235
Configuration Merging Guidelines.....	236
Data Model Merging Guidelines .....	237
Updating Data Types for Case Sensitivity .....	237
Merging Typelists – Overview .....	237
Merging Typelists – Simple Typelists .....	238
Merging Typelists – Complex Typelists.....	238
Reviewing Shared Typekey Configuration.....	238
Adding State Typelist Extensions to Jurisdiction .....	238
Merging Entity Extensions .....	239
Reviewing Custom Extensions .....	240
Reconciling the Database with Custom Extensions .....	240
Removing Obsolete Attributes .....	240
Updating Extractable Edge Foreign Keys.....	240
Converting Money to MonetaryAmount.....	240
Updating Product Model API Calls .....	241

Merging PolicyCenter Typelists . . . . .	242
GLCoverageFormType . . . . .	243
PercentDuplicated . . . . .	244
ReceptacleType . . . . .	244
GLStateCostType . . . . .	244
Upgrading the Business Auto Line Configuration . . . . .	244
Changes to the Logging API . . . . .	246
Conceptual Changes to Logging . . . . .	246
Instantiating Loggers . . . . .	247
Logging Messages . . . . .	248
Passing Loggers as Parameters . . . . .	248
Merging CADiffTree.xml and BADiffTree.xml . . . . .	249
Changes to Iterators in PCF Files . . . . .	249
Updating Namespace on Files Loaded by GX Models . . . . .	249
Merging Enhancements . . . . .	249
Updating PolicyPeriodPlugin.gs . . . . .	250
Consider Enabling Check for Small Cost Changes . . . . .	251
Merging systables.xml . . . . .	251
Merging Claim Details PCF Files . . . . .	251
Adding DDL Configuration Options to database-config.xml . . . . .	252
Merging Changes to Field Validators . . . . .	252
Renaming PCF files According to Their Modes . . . . .	252
Merging compatibility-xsd.xml . . . . .	253
Merging Display Properties . . . . .	253
Merging Other Files . . . . .	254
Fixing Gosu Issues . . . . .	254
Gosu Case Sensitivity . . . . .	254
Inequality Operator . . . . .	255
Ambiguous Method Calls . . . . .	255
Nested Comments . . . . .	256
Upgrading Rules to PolicyCenter 8.0.3 . . . . .	256
Rules Required for Free Text Search . . . . .	257
Running PCF Iterator Upgrade . . . . .	258
Translating New Display Properties and Typecodes . . . . .	258
Validating the PolicyCenter 8.0.3 Configuration . . . . .	259
Using Studio to Verify Files . . . . .	259
Starting PolicyCenter and Resolving Errors . . . . .	259
Importing Policy Forms . . . . .	260
Building and Deploying PolicyCenter 8.0.3 . . . . .	260
<b>11 Upgrading the PolicyCenter 4.0.x Database . . . . .</b>	<b>261</b>
Upgrading Administration Data for Testing . . . . .	262
Identifying Data Model Issues . . . . .	263
Verifying Batch Process and Work Queue Completion . . . . .	264
Purging Data Prior to Upgrade . . . . .	264
Purging Old Messages from the Database . . . . .	264
Purging Completed Workflows and Workflow Logs . . . . .	265
Validating the Database Schema . . . . .	265
Checking Database Consistency . . . . .	266
Creating a Data Distribution Report . . . . .	266
Generating Database Statistics . . . . .	267
Creating a Database Backup . . . . .	268
Updating Database Infrastructure . . . . .	268

Preparing the Database for Upgrade.....	268
Ensuring Adequate Free Space .....	268
Disabling Replication .....	268
Assigning Default Tablespace (Oracle only) .....	268
Deleting CoverageSymbolGroup from Coverage.....	269
Enabling Migration to 64-bit IDs (SQL Server Only) .....	269
Setting Linguistic Search Collation .....	270
Customizing the Upgrade .....	271
Running Custom Version Checks and Triggers .....	272
IDatamodelUpgrade API Examples .....	275
Upgrading Archived Entities .....	282
Disabling the Scheduler .....	284
Suspending Message Destinations .....	285
Configuring the Database Upgrade.....	285
Adjusting Commit Size for Encryption .....	285
Configuring Version Trigger Elements.....	286
Deferring Creation of Nonessential Indexes.....	287
Configuring the Upgrade on Oracle .....	288
Configuring the Upgrade on SQL Server .....	290
Downloading Database Upgrade Instrumentation Details .....	291
Checking the Database Before Upgrade.....	292
Specifying ValueType on Coverage Terms .....	292
Dropping Custom Rating Worksheet Tables .....	293
Starting the Server to Begin Automatic Database Upgrade.....	294
Test the Database Upgrade .....	294
Integrations and Starting the Server .....	295
Understanding the Automatic Database Upgrade.....	295
Version Trigger Descriptions .....	296
Viewing Detailed Database Upgrade Information .....	315
Dropping Unused Columns on Oracle .....	316
Reloading Rating Sample Data.....	317
Exporting Administration Data for Testing .....	317
Upgrading Phone Numbers.....	319
Final Steps After The Database Upgrade is Complete .....	320
Checking that Contacts Have Unique Addresses .....	320
Backing up the Database After Upgrade.....	320
<b>12 Upgrading Integrations and Gosu from 4.0.x.....</b>	<b>321</b>
Overview of Upgrading Integration Plugins and Code .....	321
Tasks Required Before Starting the Server.....	323
Tasks Required Before Deploying a Production Server.....	324
Tasks Required Before the Next Upgrade .....	324

# About PolicyCenter Documentation

The following table lists the documents in PolicyCenter documentation.

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

Document	Purpose
<i>Product Model Guide</i>	Describes the PolicyCenter product model. The intended readers are business analysts and implementation engineers who use PolicyCenter or Product Designer. To customize the product model, see the <i>Product Designer Guide</i> .
<i>Product Designer Guide</i>	Describes how to use Product Designer to configure lines of business. The intended readers are business analysts and implementation engineers who customize the product model and design new lines of business.

## Conventions in This Document

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

## Support

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

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

---

part I

# Planning the Upgrade



# Planning Your PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter upgrade to version 8.0.3. 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 35 in the *New and Changed Guide*.
- “What’s New and Changed in 7.0.0” on page 103 in the *New and Changed Guide*.
- “Release Notes Archive” on page 169 in the *New and Changed Guide* for changes to maintenance releases between major versions.
- “Upgrade Issues” in the PolicyCenter 8.0.3 release notes for issues specific to this release. If there are no upgrade issues specific to PolicyCenter 8.0.3, there is not an “Upgrade Issues” topic in the release notes.
- If you have any language packs installed, see “Upgrading Display Languages” on page 28 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 PolicyCenter 8.0, see the topics in “Upgrading from 8.0.x” on page 25.

If upgrading PolicyCenter 7.0, see the topics in “Upgrading from 7.0.x” on page 97.

If upgrading PolicyCenter 4.0, see the topics in “Upgrading from 4.0.x” on page 207.

This topic includes:

- “Supported Starting Version” on page 18
- “Upgrading Language Packs” on page 19
- “Roadmap for Planning the Upgrade” on page 19
- “Upgrade Assessment” on page 19
- “Preparing for the Upgrade” on page 21
- “Project Inception” on page 22
- “Design and Development” on page 23
- “System Test” on page 23
- “Deployment and Support” on page 23

## Supported Starting Version

You can upgrade to PolicyCenter 8.0.3 from any 7.0 version or from 4.0 versions that are 4.0.2 or newer. If you are on a 4.0 version prior to 4.0.2, first upgrade to version 4.0.2. If you are on a 3.0 version, first upgrade your database to 4.0 using the instructions in the 4.0 *Upgrade Guide*. Then proceed with the upgrade from 4.0 to 8.0 using the instructions in this guide.

### Upgrading from Version 3.0

To upgrade your database to 4.0, 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 4.0 batch server and a 4.0-compatible database. See the *Guidewire Platform Support Matrix* for 4.0 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 4.0 application will never be used interactively by any users, you do not need to upgrade your entire configuration and integrations to 4.0, only the data model.

It is not necessary to upgrade your entire custom configuration to 4.0 and then test the entire application in 4.0 before upgrading to 8.0. This process would add several months to the project duration with no significant benefit.

#### To upgrade to 4.0

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 4.0, because only the data model files must be merged. This is the minimum work necessary to be able to start the 4.0 server and trigger the automated database upgrade to 4.0. 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 28 in the *Globalization Guide* before continuing with the PolicyCenter 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 *PolicyCenter New and Changed Guide* and the release notes describe new features available after upgrading PolicyCenter. 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 PolicyCenter 8.0.3. 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 PolicyCenter 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 *PolicyCenter 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 PolicyCenter data model includes all PolicyCenter 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 PolicyCenter 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 *PolicyCenter New and Changed Guide* and *PolicyCenter 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 28 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 PolicyCenter 8.0.3 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 PolicyCenter, 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 PolicyCenter 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.



# Upgrading from 8.0.x

This part describes how to perform an upgrade from PolicyCenter 8.0.x to 8.0.3.

If you are upgrading from PolicyCenter 7.0.x, see “Upgrading from 7.0.x” on page 97 instead.

If you are upgrading from PolicyCenter 4.0.x, see “Upgrading from 4.0.x” on page 207 instead.

This part includes the following topics:

- “Upgrading the PolicyCenter 8.0.x Configuration” on page 27
- “Upgrading the PolicyCenter 8.0.x Database” on page 51
- “Upgrading PolicyCenter from 8.0.x for ContactManager” on page 91
- “Upgrading ContactManager from 8.0.x” on page 93



# Upgrading the PolicyCenter 8.0.x Configuration

This topic describes how to upgrade the PolicyCenter configuration from version 8.0.x to 8.0.3.

If you are upgrading from a 7.0.x version, see “Upgrading the PolicyCenter 7.0.x Configuration” on page 99 instead.

If you are upgrading from a 4.0.x version, see “Upgrading the PolicyCenter 4.0.x Configuration” on page 209 instead.

This topic includes:

- “Overview of ContactManager Upgrade” on page 28
- “Obtaining Configurations and Tools” on page 28
- “Creating a Configuration Backup” on page 32
- “Removing Patches” on page 32
- “Removing Language Packs” on page 32
- “Updating Infrastructure” on page 32
- “Launching the PolicyCenter 8.0.3 Configuration Upgrade Tool” on page 33
- “Configuration Upgrade Tool Automated Steps” on page 34
- “Using the PolicyCenter 8.0.3 Upgrade Tool Interface” on page 35
- “Merging Product Model Files” on page 41
- “Configuration Merging Guidelines” on page 42
- “Data Model Merging Guidelines” on page 43
- “Updating Product Model API Calls” on page 45
- “Merging Display Properties” on page 46
- “Upgrading Rules to PolicyCenter 8.0.3” on page 47
- “Translating New Display Properties and Typecodes” on page 48

- “Validating the PolicyCenter 8.0.3 Configuration” on page 48
- “Importing Policy Forms” on page 49
- “Building and Deploying PolicyCenter 8.0.3” on page 50

## Overview of ContactManager Upgrade

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

- For information on upgrading ContactManager, see “Upgrading ContactManager from 8.0.x” on page 93.
- For information on upgrading PolicyCenter files used to integrate with ContactManager, see “Upgrading PolicyCenter from 8.0.x for ContactManager” on page 91.

## 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter version.

**Target** – The unedited, original configuration of PolicyCenter 8.0.3 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 PolicyCenter 8.0.3 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.

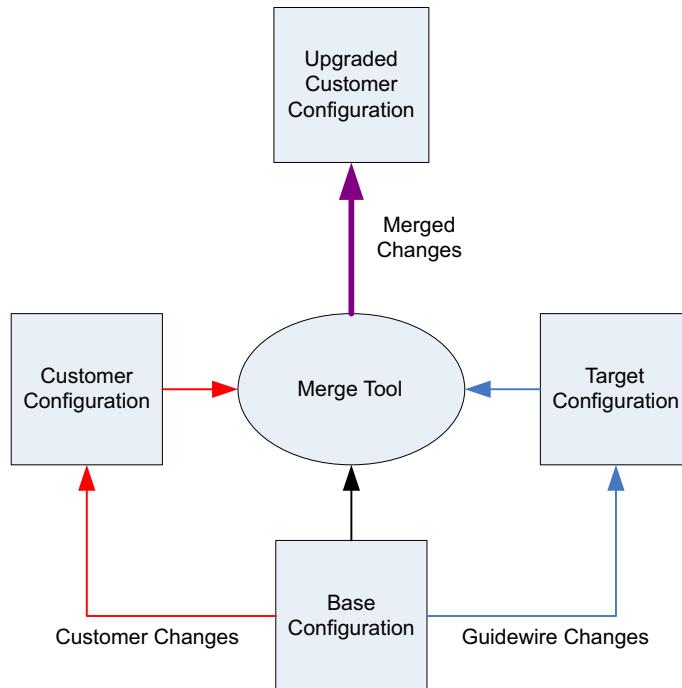
---

**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 PolicyCenter.
5. Click Upgrade Diff Reports - PolicyCenter or Upgrade Diff Reports - ContactManager.
6. Click Upgrade From *base version*.
7. Click Upgrade To 8.0.3.

## Specifying Configuration and Tool Locations

The PolicyCenter 8.0.3 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 3 Professional. 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 PolicyCenter 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 `PolicyCenter/modules/ant/upgrade.properties` file of the target PolicyCenter 8.0.3 environment. Remove the pound sign, '#', preceding each property to uncomment the line and then specify values. Use double backslashes in paths. For example, `C:\\\\PolicyCenter`. 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 PolicyCenter directory of the current customer environment. This directory contains <code>/bin</code> and other PolicyCenter 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 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> .
<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. 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 display order, from left to right, for versions of a file viewed in the difference editor tool specified by <code>upgrader.merge.tool</code> . By default, the tool displays, left to right, the versions of a file in this order:  <code>NewFile OldFile ConfigFile</code> in which:  <code>NewFile</code> is the unmodified target version provided with PolicyCenter 8.0.3. <code>OldFile</code> is the original base version. <code>ConfigFile</code> is your configured version.  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:  <b>Araxis Merge Professional</b> <code>upgrader.merge.tool.arg.order = NewFile OldFile ConfigFile</code> <b>Beyond Compare 3 Professional</b> <code>upgrader.merge.tool.arg.order = NewFile ConfigFile OldFile</code> <b>P4Merge</b> <code>upgrader.merge.tool.arg.order = OldFile NewFile ConfigFile</code> You might need to configure the display of your merge tool to show three panels.
<code>upgrader.steps.class</code>	The class to run to execute the configuration upgrade automated steps. If you are upgrading PolicyCenter 4.0 or newer, then leave this property commented out.
<code>exclude.pattern</code>	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.

## 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.

### Backing up the Configuration

Guidewire recommends that you track PolicyCenter configuration changes in a source code control system. Before upgrading, have a labeled version of your entire pre-upgrade PolicyCenter 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.

Before upgrading PolicyCenter, commit all changes in all open Product Designer change lists. Uncommitted changes are discarded during the upgrade process.

### Backing up the Product Model

Normally, backing up the existing `config` directory backs up the product model. You can back it up separately by saving a copy of the `config/resources/productmodel` directory.

## Removing Patches

If you have applied any patches from Guidewire to PolicyCenter, remove the patches before you run the configuration upgrade. Patches are specific to the pre-upgrade version. If you 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 PolicyCenter. See “Upgrading Display Languages” on page 28 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 PolicyCenter 8.0.3 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 101.

Also see the *PolicyCenter 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.

### Copying Custom Rules and Adding PolicyCenter 8.0.3 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 PolicyCenter 8.0.3 to a PolicyCenter 8.0.3 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 Parameter

The upgrade updates the `config.xml` file to rename the parameter `WorksheetContainerAgeForPurging` to `RatingWorksheetContainerAgeForPurging`, if the parameter is in `config.xml`.

### Updating Product Model Structure

The upgrade updates the product model directory structure within `config/resources/productmodel`.

- State-specific elements are moved to state-specific configuration files within state-specific subdirectories.
- All policy line patterns and associated lookup files are split according to their line of business (LOB) and moved into a LOB-specific `LOB_name` directory.
- All clause patterns and associated lookup files in the base PolicyCenter configuration are split up into LOB-specific `LOB_name/coveragepatterns` directories.
- All images are moved into LOB-specific `LOB_name/images` directories.
- Properties files are moved to a LOB-specific `LOB_name` directory.
- Clauses are directly linked to their associated policy line in the clause file. Coverage category information is still included but now becomes only an attribute of the clause, not the way it is linked to the policy line.
- Grandfathering information for offerings, modifiers, clauses, coverage terms and coverage term options is moved into separate files named `LOB_nameLine-grandfathering`.
  - Base information is located in the `LOB_nameLine.xml` file in the `LOB_name` directory.
  - Non-jurisdiction-specific grandfathering information is moved to a `LOB_nameLine-grandfathering` in the `LOB_name` directory.
  - Jurisdiction-specific grandfathering information is moved to a `LOB_nameLine-grandfathering` in the `LOB_name/jurisdictions/jurisdictionName` directory.
- State-specific and non-state-specific modifier minimum and maximum information are moved into state-specific and non-state-specific files.

- Product model lookup files for policy lines, clause, products, offerings and question sets are broken up into non-jurisdiction specific and jurisdiction-specific files and are placed in appropriate directories.
- Product pattern and associated lookup files in the base Policy Center configuration are broken up into product-specific *productName* directories.
- Offerings and associated look-up files in the base Policy Center configuration are broken up into product-specific *productName/offerings* directories.
- Product image files in the base Policy Center configuration are broken up into product-specific directories.

## Using the PolicyCenter 8.0.3 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:

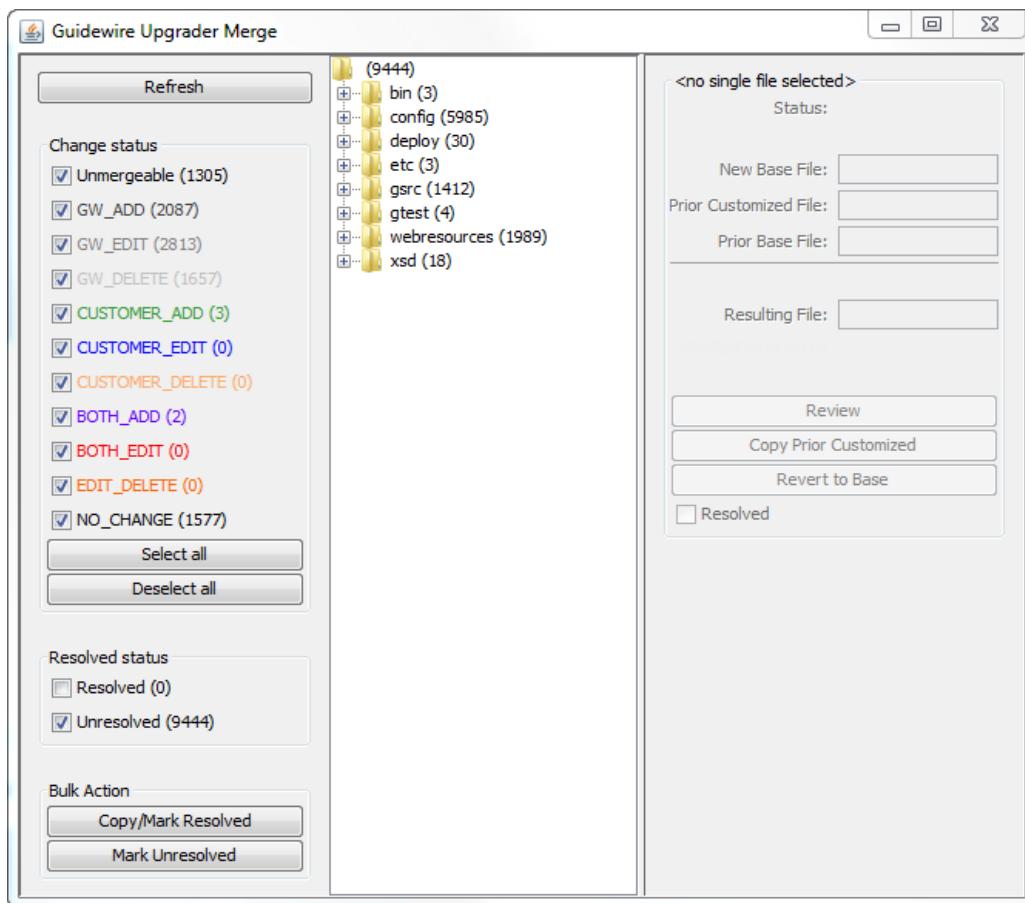
- The *customer* file.
- The *base* file, from which you configured the customer file.
- The *target* file, from PolicyCenter 8.0.3.

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 37.
- 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 and product model 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.  Typically, you use exclude.pattern to specify source control metadata files. Samples are provided in upgrade.properties for CVS and SVN.
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 <b>Delete</b> , the Configuration Upgrade Tool removes the file from the target configuration. If you click <b>Revert to Base</b> , 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 <b>Copy prior customized</b> to move the file to the target configuration.</p> <p>The <b>EDIT_DELETE</b> 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 PolicyCenter. 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>PolicyCenter 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 <b>CUSTOMER_EDIT</b> 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 PolicyCenter 8.0.3.

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 37.
- **New base file** – The new version of this file supplied by Guidewire with PolicyCenter 8.0.3. 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 PolicyCenter 8.0.3 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.

## Merging Product Model Files

Do not follow the normal upgrade process when working with product model files. The base product model content is just a starting point for custom configurations. Any modifications that Guidewire makes to the base content will already have been made to your product model if it was relevant to your business. Therefore, do not attempt to merge in content changes to the base product model files into your versions of the files. The only changes to take are syntactic changes, for which there are configuration upgrade triggers. Consequently, product model files are labelled `Unmergeable` by the Configuration Upgrade Tool.

**The normal process to merge configuration files is:**

1. Open the Configuration Upgrade Tool.
2. Select a file.
3. Merge versions of the file.
4. Save the merged file in the default location. The merged file must have the same name as the original file being resolved.
5. Close the merge tool.
6. Answer Yes to the copy question when the Configuration Upgrade Tool prompts you.

However, product model files, once configured, represent a carrier's insurance policies. Do not merge product model files with updated content delivered by Guidewire. Only update your configuration with automated syntax changes to the product model files. These syntax changes are made by automated steps of the Configuration Upgrade Tool.

In addition to the following procedure, many other files that must be merged have dependencies upon the product model of the carrier. Only merge changes by Guidewire to these files if they do not conflict with the existing product model. Some examples of such files are:

- config/lookuptables/lookuptables.xml
- Policyline-specific Gosu enhancements
- Rating Gosu classes
- Policyline-specific validation Gosu classes

**Instead, the product model-specific process is:**

1. Open the Configuration Upgrade Tool. Doing this executes automated syntax upgrades to product model files.
2. Close the Configuration Upgrade Tool.
3. Delete all of the PolicyCenter 8.0 default product model files by deleting the config/resources/productmodel folder.
4. Manually copy all files in the temporary upgrade config/resources/productmodel directory to your new configuration directory.
5. Manually copy config/locale/\*/productmodel.display.properties for each defined locale from the temporary upgrade directory to your new configuration module. If you used unmodified product model patterns in PolicyCenter 4.0, then add the contents of productmodel.display.properties from the PolicyCenter 4.0 pc module to the productmodel.display.properties in the new configuration module.

## Configuration Merging Guidelines

The first milestone of an upgrade project is to generate the Java and SOAP APIs (by running gwpc regen-java-api and gwpc 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 101.

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, PolicyCenter 8.0.3. 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, PolicyCenter enforces restrictions on the use of shared typekeys among subtypes.

### Same Field Name and Typelist with Different Column

In PolicyCenter 7.0 and earlier, if a shared typekey had the same field name and typelist, and specified a different column name, PolicyCenter created only one of the typekey columns. The shared typekeys were stored in the single column. As of PolicyCenter 8.0.3, if a shared typekey with the same field name and typelist specifies a different column name, PolicyCenter 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 PolicyCenter 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 PolicyCenter 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

PolicyCenter 8.0.3 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 PolicyCenter 8.0.3. PolicyCenter 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 PolicyCenter.

### To generate the data dictionary

1. From the command line, navigate to the `bin` directory of the target version.
2. Run the command `gwpc 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.

## Updating Product Model API Calls

In PolicyCenter 8.0.1 and newer, Guidewire updated several product model API classes to implement a public interface rather than extend a public abstract class. This change simplifies the API and prevents potentially dangerous methods from being exposed. The public interfaces do not include a `getByCode` method that was available on the abstract classes. Instead, a related lookup class provides the method to retrieve the product model object. All of the new interfaces are within the `gw.api.productmodel` package.

For upgrades from versions prior to 8.0.1, update your code to change calls to the `getByCode` method to the new method available on the lookup class. The new method is provided in the following table.

Old method	New method
<code>ChoiceCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
<code>ChoiceCovTermPattern</code> extended the public abstract class <code>CovTermPatternInternal</code> .	<code>ChoiceCovTermPattern</code> extends the public interface <code>CovTermPattern</code> .
<code>ClausePattern.getByCode(code)</code>	<code>ClausePatternLookup.getByCode(code)</code>

Old method	New method
<code>ConditionPattern.getByCode(code)</code>	<code>ClausePatternLookup.getConditionPatternByCode(code)</code>
ConditionPattern extended the public abstract class <code>ClausePattern</code> .	ConditionPattern extends the public interface <code>ClausePattern</code> .
<code>CoveragePattern.getByCode(code)</code>	<code>ClausePatternLookup.getCoveragePatternByCode(code)</code>
CoveragePattern extended the public abstract class <code>ClausePattern</code> .	CoveragePattern extends the public interface <code>ClausePattern</code> .
<code>CovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
<code>CoverageCategory.getByCode(code)</code>	<code>CoverageCategoryLookup.getByCode(code)</code>
<code>CoverageSymbolPattern.getByCode(code)</code>	<code>CoverageSymbolPatternLookup.getByCode(code)</code>
<code>DirectCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
DirectCovTermPattern extended the public abstract class <code>CovTermPatternInternal</code> .	DirectCovTermPattern extends the public interface <code>CovTermPattern</code> .
<code>ExclusionPattern.getByCode(code)</code>	<code>ClausePatternLookup.getExclusionPatternByCode(code)</code>
ExclusionPattern extended the public abstract class <code>ClausePattern</code> .	ExclusionPattern extends the public interface <code>ClausePattern</code> .
<code>ModifierPatternBase.getByCode(code)</code>	<code>ModifierPatternBaseLookup.getByCode(code)</code>
<code>Offering.getByCode(code)</code>	<code>OfferingLookup.getByCode(code)</code>
<code>OptionCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
OptionCovTermPattern extended the public abstract class <code>ChoiceCovTermPattern</code> , which in turn extended the public abstract class <code>CovTermPatternInternal</code> .	OptionCovTermPattern extends the public interface <code>ChoiceCovTermPattern</code> , which in turn extends the public interface <code>CovTermPattern</code> .
<code>PackageCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
PackageCovTermPattern extended the public abstract class <code>ChoiceCovTermPattern</code> , which in turn extended the public abstract class <code>CovTermPatternInternal</code> .	PackageCovTermPattern extends the public interface <code>ChoiceCovTermPattern</code> , which in turn extends the public interface <code>CovTermPattern</code> .
<code>PolicyLinePattern.getByCode(code)</code>	<code>PolicyLinePatternLookup.getByCode(code)</code>
<code>Product.getByCode(code)</code>	<code>ProductLookup.getByCode(code)</code>
<code>Question.getByCode(code)</code>	<code>QuestionLookup.getByCode(code)</code>
<code>QuestionChoice.getByCode(code)</code>	<code>QuestionChoiceLookup.getByCode(code)</code>
<code>QuestionSet.getByCode(code)</code>	<code>QuestionSetLookup.getByCode(code)</code>
<code>RateFactorPatternBase.getByCode(code)</code>	<code>RateFactorPatternBaseLookup.getByCode(code)</code>
<code>TypekeyCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
TypekeyCovTermPattern extended the public abstract class <code>CovTermPatternInternal</code> .	TypekeyCovTermPattern extends the public interface <code>CovTermPattern</code> .

See the *Upgrade Diffs* report for more API changes.

## 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 PolicyCenter 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 PolicyCenter 8.0.3 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 48.

In PolicyCenter 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 PolicyCenter 8.0.3

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 PolicyCenter 8.0.3 Rules folder within `modules/configuration/config/rules`. Use Studio to update your rules. You can use the rules in the PolicyCenter 8.0.3 folder as a comparison. Compare your custom rules to the new default 8.0.3 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 8.0.3 versions to determine what types of changes Guidewire has made.

### To compare rules between versions using the Rule Repository Report

5. 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 PolicyCenter directory.  
If you want to compare your custom rules against the PolicyCenter 8.0.3 rules, do not move the `modules/configuration/config/rules` directory from your starting version. However, do remove the `PolicyCenter<base version>` directory from `modules/configuration/config/rules/rules` of the starting version if this directory exists.
6. Open a command window.
7. Navigate to the `bin` directory of your starting version.
8. Enter the following command:  
`gwpc regen-rulereport`  
This command creates a rule repository report XML file in `build/rules`.
9. Append the starting version number to the XML file name.
10. Restore moved directories to the starting version.
11. Install files for a fresh PolicyCenter 8.0.3 version. This is a separate configuration from the target configuration that you have merged. This version will only contain the default rules provided with PolicyCenter 8.0.3.
12. Navigate to the `bin` directory of the new PolicyCenter 8.0.3 version.
13. Enter the following command:  
`gwpc 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.

14. Append the target version number to the XML file name.
15. 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 PolicyCenter 8.0.3 rules by opening the default rules in the PolicyCenter 8.0.3 directory within `configuration → config → Rule Sets`. When you have finished updating all of your custom rules, delete the PolicyCenter 8.0.3 rules directory from `modules/configuration/config/rules`.

## Translating New Display Properties and Typecodes

PolicyCenter 8.0.3 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 PolicyCenter environment, skip this procedure.

### To localize new display properties and typecodes

1. Export display keys by running the following command from your PolicyCenter 8.0.3 environment `PolicyCenter/bin` directory:  
`gwpc 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 PolicyCenter 8.0.3 environment `PolicyCenter/bin` directory:  
`gwpc 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 PolicyCenter 8.0.3 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 PolicyCenter. Do not start PolicyCenter at this point. Studio can run without connecting to the application server.

**To validate Studio resources**

1. Start Guidewire Studio by running `gwpc studio` from the `PolicyCenter\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 PolicyCenter and Resolving Errors

**IMPORTANT** In the process described in this section, do not point the PolicyCenter server at a production database. The goal of this process is to test the configuration upgrade. Create an empty database account and point PolicyCenter to this account for this process. See “Configuring the Database” on page 27 in the *Installation Guide* and “Deploying PolicyCenter to the Application Server” on page 82 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.  
PolicyCenter 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.

## Importing Policy Forms

1. Start your development server by opening a command window to `PolicyCenter/bin` and running the `gwpc dev-start` command.
2. Log in to the development server as `su`.
3. Click the **Administration** main tab.
4. Click **Import/Export Data** in the left navigation area.
5. Import the file `modules/configuration/policy_forms.xml` from the temporary upgrade directory.

# Building and Deploying PolicyCenter 8.0.3

After you apply and validate an upgrade to the configuration environment, rebuild and redeploy PolicyCenter. 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 PolicyCenter to the application server. See “Deploying PolicyCenter to the Application Server” on page 82 in the *Installation Guide* of the target version for instructions.

---

**WARNING** Do not yet start PolicyCenter. Only package the application file and deploy it to the application server. Starting PolicyCenter 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 PolicyCenter. JAR files in this location survive the upgrade process.

# Upgrading the PolicyCenter 8.0.x Database

This topic provides instructions for upgrading the PolicyCenter database to PolicyCenter 8.0.3.

If you are upgrading from a 7.0.x version, see “Upgrading the PolicyCenter 7.0.x Database” on page 143 instead.

If you are upgrading from a 4.0.x version, see “Upgrading the PolicyCenter 4.0.x Database” on page 261 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
- “Customizing the Upgrade” on page 60
- “Disabling the Scheduler” on page 73
- “Suspending Message Destinations” on page 74
- “Configuring the Database Upgrade” on page 74
- “Checking the Database Before Upgrade” on page 81
- “Starting the Server to Begin Automatic Database Upgrade” on page 81

- “Viewing Detailed Database Upgrade Information” on page 85
- “Dropping Unused Columns on Oracle” on page 86
- “Reloading Rating Sample Data” on page 87
- “Exporting Administration Data for Testing” on page 87
- “Final Steps After The Database Upgrade is Complete” on page 89

## Upgrading Administration Data for Testing

You might want to create an upgraded administration data set for development and testing of rules and libraries with PolicyCenter 8.0.3. 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 87. 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) PolicyCenter production instance:
  - a. Log on to PolicyCenter 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**. PolicyCenter 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 `PolicyCenter/bin` and entering the following command:  
`gwpc dev-start`
5. Import this administration data into the development environment.
  - a. Log on to PolicyCenter 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 PolicyCenter 8.0.3. 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 27 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 PolicyCenter 8.0.3 configuration to the restored database.
10. Start the PolicyCenter 8.0.3 server to upgrade the database.
11. Export the upgraded administration data:
  - a. Start the PolicyCenter 8.0.3 server by navigating to `PolicyCenter/bin` and running the following command:  
`gwpc dev-start`
  - b. Open a browser to PolicyCenter 8.0.3.
  - 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 PolicyCenter 8.0.3.

## 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 89.

### 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 PolicyCenter installation at an empty schema and starting the PolicyCenter server. See “Deploying PolicyCenter to the Application Server” on page 82 in the *Installation Guide*.
2. Complete the configuration upgrade for data model files in your starting version, according to the instructions in “Upgrading the PolicyCenter 8.0.x Configuration” on page 27. 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 66 in the *Installation Guide*.
4. Start the PolicyCenter server in your upgraded development environment. The server performs the database upgrade to PolicyCenter 8.0.3. See “Starting the Server to Begin Automatic Database Upgrade” on page 81.
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 PolicyCenter as the superuser.
2. Press Alt + Shift + T. PolicyCenter 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 PolicyCenter.

### 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 `pc_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 PolicyCenter, 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 `pc_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 PolicyCenter creates an activity, the activity is added to the `pc_Workflow`, `pc_WorkflowLog` and `pc_WorkflowWorkItem` tables. Once a user completes the activity, PolicyCenter sets the workflow status to completed. The `pc_Workflow`, `pc_WorkflowLog` and `pc_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.

PolicyCenter 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 `PolicyCenter/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 `PolicyCenter/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

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

### To run consistency checks

1. Start the PolicyCenter server if it is not already running.
2. Log in to PolicyCenter 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**.

For more information about the **Consistency Checks** page, see “**Consistency Checks**” on page 147 in the *System Administration Guide*.

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 PolicyCenter **Consistency Checks** page.

## 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 PolicyCenter 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 65 in the *System Administration Guide*.

### To create a database distribution report

1. In `config.xml`, set `<param name="EnableInternalDebugTools" value="true"/>`.
2. Start the PolicyCenter application server.

3. Log into PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter database.

You can configure SQL Server to periodically update statistics using SQL. See your database documentation and “Configuring Database Statistics” on page 42 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 44 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 PolicyCenter 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 PolicyCenter 8.0.3. 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 PolicyCenter. The only exception is when the PolicyCenter 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 PolicyCenter to collate search results. The `strength` attribute of the `LinguisticSearchCollation` element of `GWLanguage` in `language.xml` in the currently selected region specifies how PolicyCenter sorts search results. You can set the `strength` to `primary` or `secondary`.

With `LinguisticSearchCollation strength` set to `primary`, PolicyCenter searches results in a case-insensitive and accent-insensitive manner. PolicyCenter 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`, PolicyCenter treats “Renée”, “Renée”, “renée” and “reneé” the same.

With `LinguisticSearchCollation strength` set to `secondary`, PolicyCenter searches results in a case-insensitive, accent-sensitive manner. PolicyCenter 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 PolicyCenter search treats “Renée” and “renée” the same but treats “Renée” and “reneé” differently. By default, PolicyCenter 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`.

PolicyCenter 7.0 introduced configurable linguistic searching for SQL Server databases. In releases prior to PolicyCenter 7.0, PolicyCenter 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”.

PolicyCenter populates denormalized values of searchable columns to support the search collation. For example, with `LinguisticSearchCollation strength` set to `primary`, PolicyCenter stores the value “Renée”, “Renée”, “reneé” and “reneé” in a denormalized column as “reneé”. With `LinguisticSearchCollation strength` set to `secondary`, PolicyCenter stores a denormalized value of “renée” for “Renée” or “reneé” 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, PolicyCenter repopulates the denormalized columns. Previous versions of PolicyCenter 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`, PolicyCenter 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 PolicyCenter database upgrade from a prior major version repopulates the denormalized columns regardless of the `LinguisticSearchCollation strength` value. PolicyCenter 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 157 in the *Globalization 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** PolicyCenter 4.0 included a similar plugin interface, `IDatabaseUpgrade`. If you previously implemented `IDatabaseUpgrade` for an upgrade to PolicyCenter 4.0, 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 `gwpc regen-gosudoc` command from the PolicyCenter `bin` directory. Then, open `PolicyCenter/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 64.

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

- a. Start Guidewire Studio 8.0.3 by entering `gwpc studio` from the `PolicyCenter/bin` directory.
- b. In Studio, expand `configuration` → `config` → `Plugins`.
- c. Right-click `registry` and click `New` → `Plugin`.
- d. In the `Plugin` dialog, enter a name, such as `DatamodelUpgradePlugin`.
- 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()
```

## Upgrading Archived Entities

If you implement archiving, and you make custom data model changes, then you can upgrade the archived XML using the `IDatamodelUpgrade` plugin.

Simple data model changes do not require a custom trigger. These include:

- Adding a new entity
- Updating denormalization columns
- Adding editable columns such as `updatetime`
- Adding new columns
- Changing the nullability of a column

More complex transformations or those that could result in loss of data require a version trigger. These include:

- changing a datatype (other than just length)
- migrating data from one table or column to another
- dropping a column
- dropping a table
- renaming a column
- renaming a table

PolicyCenter upgrades an archived entity as the entity is restored.

The `IDatamodelChange` interface includes a `getArchivedDocumentUpgradeVersionTrigger` method that returns an `ArchivedDocumentUpgradeVersionTrigger`.

You can define custom `ArchivedDocumentUpgradeVersionTrigger` entities to modify archived XML. The `ArchivedDocumentUpgradeVersionTrigger` is an abstract class that you can extend to create your custom triggers.

Define the constructor of your custom `ArchivedDocumentUpgradeVersionTrigger` to call the constructor of the superclass and pass it a numeric value. For example:

```
construct() {
    super(171)
}
```

This numeric value is the extension version to which your trigger applies. If you run the upgrade against a database with a lower extension version, then your custom trigger is called. The current extension version is defined in `modules/configuration/config/extensions/extensions.properties`.

Provide an override definition of the `get Description` property to return a `String` that describes the actions of your trigger.

Provide an override definition for the `execute` function to define the actions that you want your custom trigger to make on archived XML.

When the upgrade executes your custom trigger, it wraps each XML entity in an `IArchivedEntity` object. Each typekey is wrapped in an `IArchivedTypekey` object. The upgrade operates on a single XML document at a time.

`ArchivedDocumentUpgradeVersionTrigger` provides the following key operations:

- `getArchivedEntitySet(entityName : String)` – returns an `IArchivedEntitySet` object that contains all `IArchivedEntity` objects of the given type in the XML document.

`IArchivedEntitySet` provides the following key methods:

- `rename(newEntityName : String)` – renames all rows in the set to the new name.
- `delete()` – deletes all rows in the set.
- `search(predicate : Predicate<IArchivedEntity>)` – returns a `List` of `IArchivedEntity` objects that match the given predicate.
- `create(referenceInfo : String, properties : List<Pair<String, Object>>)` – returns a new `IArchivedEntity` with the given properties.

`IArchivedEntity` provides the following key methods:

- `delete()` – deletes just this row.
- `getPropertyValue(propertyName : String)` – returns the value of the property of the given name. If the property value is a reference to another entity, this method returns an `IArchivedEntity`.
- `move(newEntityName : String)` – moves this to a new entity type of the given name. The type is created if it did not exist. You must add any required properties to the type.
- `updatePropertyValue(propertyName : String, newValue : String)` – updates the property of the given name to the given value.
- `getArchivedTypekeySet(typekeyName : String)` – returns an `IArchivedTypekeySet` object that contains all `IArchivedTypekey` objects in the given typelist.
- `getArchivedTypekey(typelistName : String, code : String)` – Returns an `IArchivedTypekey` representing the typekey.

`IArchivedTypekey` provides the following key method:

- `delete()` – deletes the typekey from the XML.

More methods are available for `IArchivedEntitySet`, `IArchivedEntity` and `IArchivedTypekey`. See the Gosu documentation for a full listing. Generate the Gosu documentation by navigating to the PolicyCenter bin directory and entering the following command:

```
gwpc regen-gosudoc
```

## Incremental Upgrade

When PolicyCenter archives an entity, it records the current data model version on the entity. PolicyCenter upgrades an archived entity as the entity is restored. The upgrader executes the necessary archive upgrade triggers incrementally on each archived XML entity, according to the data model version of the archived entity.

An entity archived at one version might not be restored and upgraded until several intermediate data model upgrades have been performed. Therefore, do not delete your custom upgrade triggers.

Consider the following situation. Note that this example is for demonstration purposes only. The version numbers included do not represent actual PolicyCenter versions but are included to explain the incremental upgrade process. Each '+' after a version number indicates a custom data model change.

#### **Guidewire data model changes**

v6.0.0 – Entity does not have column X.

v6.0.1 – Guidewire adds column X to the entity with a default value of 100.

v6.0.2 – Guidewire updates the default value of column X to 200.

#### **Implementation #1 upgrade path**

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.0+++ – More custom data model changes.

v6.0.2+ – column X introduced with a default value of 200.

Result of upgrade of an entity archived at v6.0.0+: column X has value 200.

In this situation, version 6.0.1 was skipped in the upgrade path. Therefore, column X is added with the default value of 200 that it has in version 6.0.2.

#### **Implementation #2 upgrade path**

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.1+ – column X introduced with a default value of 100.

v6.0.2+ – column X already exists.

Result of upgrade of an entity archived at v6.0.0+: column X has value 100.

In this situation, column X is added in version 6.0.1 with the default value of 100. During the upgrade from version 6.0.1 to version 6.0.2, column X already exists. Therefore, the upgrader does not add the column. A custom trigger would be required to update the value of X from 100 to 200 during the upgrade from version 6.0.1 to 6.0.2.

## 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 PolicyCenter 8.0.3 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 PolicyCenter from sending messages until you have verified a successful database upgrade.

### To suspend message destinations

1. Start the PolicyCenter server for the pre-upgrade version.
2. Log in to PolicyCenter 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.

## Configuring the Database Upgrade

You can set parameters for the database upgrade in the PolicyCenter 8.0.3 `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 PolicyCenter to the Application Server” on page 82 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 PolicyCenter encryption is applied on one or more attributes, the PolicyCenter 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, PolicyCenter:

- 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 77.
<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 PolicyCenter 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 PolicyCenter retrieves the current state of the counters at the end of the execution of the version trigger. PolicyCenter 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 and indexes on the `ArchivePartition` column on all entities that PolicyCenter can archive. 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, PolicyCenter runs the `DeferredUpgradeTasks` work queue as soon as the upgrade completes and the server is completely started. The `DeferredUpgradeTasks` work queue creates the nonessential performance indexes and indexes on archived entities. The database user must have permission to create indexes until after the `DeferredUpgradeTasks` work queue is complete.

Deferring nonessential index creation can shorten the duration of the upgrade process. The PolicyCenter 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` work queue 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` work queue has run to completion, PolicyCenter 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.

Do not use the archiving feature until the `DeferredUpgradeTasks` work queue has completed successfully.

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

If you do not opt to defer creation of nonessential indexes, PolicyCenter 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 PolicyCenter database upgrade marks columns as unused.

To configure the PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter does not generate statistics. This setting enables the upgrade to update statistics on changed objects. It also configures PolicyCenter 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, PolicyCenter 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 PolicyCenter 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 44 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 `pc_someTable_EXT`, specify a `<tablestatistics>` element for that table with the `action` attribute set to `keep`:

```
<database>
  ...
  <databasestatistics>
    <tablestatistics name="pc_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. 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 tempbldb 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 85.

## 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 PolicyCenter and you can also add custom version checks.

You can configure PolicyCenter 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 PolicyCenter 8.0.3 by running the following command from the bin directory:

```
gwpc 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, PolicyCenter 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.

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

If PolicyCenter 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, PolicyCenter runs all version checks regardless of a failure in one of the checks. During a regular upgrade, PolicyCenter 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.

## 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 PolicyCenter 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 PolicyCenter 8.0.3. 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.

---

Guidewire requires that you use a separate mirror database for reporting. If you did not do so, then you can experience problems during a database upgrade that are severe enough to prevent the upgrade.

In particular, the Premium Accounting extension package SQL scripts create special tables in the reporting database that reporting uses for storing premium accounting accrual data. The reporting scripts use these tables, but the PolicyCenter application does not. If you created these tables in a production database, then any attempt to upgrade that production database will fail.

If you encounter this situation, move data from all `pcrt_` prefixed tables to another schema. Then, drop all `pcrt_` prefixed tables from the database that you want to upgrade before starting the server to launch the upgrade.

## 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 *PolicyCenter 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 PolicyCenter 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 pc\_Dynamic\_Assign

The upgrade truncates the `pc_Dynamic_Assign` table.

### Dropping pc\_t1\_Template

The upgrade drops the `pc_t1_Template` table.

### Dropping Columns from WorkItem Tables

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

### Dropping Contact.CityKanjiDenorm

The upgrade drops the `pc_Contact.CityKanjiDenorm` column if it exists.

### Creating AgencyBillPlan Records

For each record in `pc_Organization` with a non-null `AgencyBillPlanID`, the upgrade creates a record in `pc_AgencyBillPlan` with the organization ID and `AgencyBillPlanID`.

### Dropping Rating Worksheet Tables

The upgrade drops tables and entities that store `RatingWorksheets` directly. As of PolicyCenter 8.0.1, rating worksheets are stored on a `WorksheetContainer` stored on a policy. The `RatingWorksheet` delegate has been removed.

### Moving CommissionPlanID from ProducerCode to ProducerCodeCurrency

The upgrade copies the `pc_ProducerCode.CommissionPlanID` to `pc_ProducerCodeCurrency.CommissionPlanID` and then deletes `pc_ProducerCode.CommissionPlanID`.

## Deleting Checksums for Product Model Lookups

The upgrade deletes checksums for product model lookups from the database. This forces PolicyCenter to resynchronize the database with product model files.

## Moving PolicyPeriod.NewInvoiceStream.Selected to PolicyPeriod.CustomBilling

The upgrade moves and renames the `Selected` column from `PolicyPeriod.NewInvoiceStream.Selected` to `PolicyPeriod.CustomBilling`. The `Selected` column did not indicate that the user selected to send a new invoice stream to BillingCenter. Instead, the user indicated custom billing and PolicyCenter either sends an invoice stream or modifies an existing one. PolicyCenter did not send any invoice stream information otherwise. Therefore the column has been renamed to `PolicyPeriod.CustomBilling` to better reflect its purpose.

## Creating the SelectedPaymentPlan PaymentPlanSummary

In PolicyCenter 8.0.2, the relationship between `PolicyPeriod` and `PaymentPlanSummary` was changed to a one-to-one relationship. The upgrade first checks that each non-retired `PaymentPlanSummary` record has a unique combination of `PolicyPeriod` and `BillingId`. The upgrade reports an error if it finds `PaymentPlanSummary` records with matching `PolicyPeriod` and `BillingId`. If the upgrade reports this error, either retire or remove the duplicate rows and restart the upgrade. When a record is restored from the archive it is upgraded to the current version. If a duplicate `PaymentPlanSummary` is detected during restoration, PolicyCenter marks the duplicate `PaymentPlanSummary` as retired.

If the upgrade finds no errors, it creates a new `SelectedPaymentPlan PaymentPlanSummary` for all `PolicyPeriods` and removes obsolete `PaymentPlanSummary` types.

## Deleting Checksums for Product Model Lookups and Lookup Rows

PolicyCenter 8.0.3 has a restructured product model. The upgrade deletes checksums for product model lookups and lookup rows from the database to allow resynchronization of product model files when the server starts.

The upgrade deletes all rows from the following tables:

- `pc_CondLookup`
- `pc_CovLookup`
- `pc_CovTermLookup`
- `pc_CovTermOptLookup`
- `pc_CovTermPackLookup`
- `pc_ExclLookup`
- `pc_ModifierLookup`
- `pc_OfferingLookup`
- `pc_ProductLookup`
- `pc_ProductModifierLookup`
- `pc_ProdRateFactorLookup`
- `pc_RatingFactorLookup`
- `pc_QuestionLookup`
- `pc_QuestionSetLookup`

# Viewing Detailed Database Upgrade Information

PolicyCenter 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 PolicyCenter server if it is not already running.
2. Log in to PolicyCenter 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 PolicyCenter 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. PolicyCenter 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
```

## Reloading Rating Sample Data

If you are using any rating data provided by Guidewire, such as calcRoutines, rateBooks, rateTableDefinition, parameter sets, and so forth, remove all existing rating data, and reload new rating data.

#### To reload rating sample data

1. Start the PolicyCenter server.
2. Remove the old rating sample data by running the following Gosu script against the database:

```
uses gw.transaction.Transaction
uses gw.api.database.Query

function findEntity<T extends KeyableBean>() : List<T>{
    var q = Query.make(T)
    q.startsWith("PublicID", "pc:", false /*ignoreCase*/)
    return q.select().toList()
}

Transaction.runWithNewBundle(\ bundle -> {
    findEntity<RateBook>().each(\ rb -> bundle.add(rb).remove())
    findEntity<RateTableDefinition>().each(\ rt -> bundle.add(rt).remove())
    findEntity<RateTableMatchOpDefinition>().each(\ rb -> bundle.add(rb).remove())
    findEntity<RateFactorRow>().each(\ rb -> bundle.add(rb).remove())
    findEntity<CoverageRateFactor>().each(\ rb -> bundle.add(rb).remove())
    findEntity<CalcRoutineDefinition>().each(\ rb -> bundle.add(rb).remove())
    findEntity<CalcRoutineParameterSet>().each(\ rb -> bundle.add(rb).remove())
}, "su")
```

3. Run the following Gosu script to reload the PolicyCenter 8.0.3 rating sample data:

```
uses gw.transaction.Transaction
uses gw.sampledata.small.SmallSampleRatingData
uses gw.sampledata.tiny.TinySampleRatingData

Transaction.runWithNewBundle(\ bundle -> {
    var tinyData = new TinySampleRatingData()
    tinyData.load()
    var sampleData = new SmallSampleRatingData()
    sampleData.load()
}, "su")
```

## 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 PolicyCenter 8.0.3. 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 PolicyCenter 8.0.3 server by navigating to `PolicyCenter/bin` and running the following command:  
`gwpc dev-start`
  - b. Open a browser to PolicyCenter 8.0.3.
  - 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 PolicyCenter 8.0.3. 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 27 in the *Installation Guide* for instructions to configure the database account.
3. Install a new PolicyCenter 8.0.3 development environment. Connect this development environment to the new database account that you created in step 2. See the *PolicyCenter 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 PolicyCenter 8.0.3 development server by navigating to `PolicyCenter/bin` and running the following command:  
`gwpc dev-start`
  - b. Open a browser to PolicyCenter 8.0.3.
  - 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 89
- “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 PolicyCenter in a production environment.
- “Completing Deferred Upgrade” on page 89
- “Backing up the Database After Upgrade” on page 89

### Checking that Contacts Have Unique Addresses

An Address cannot be shared by more than one Contact. PolicyCenter 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 PolicyCenter 8.0.3 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 work queue as soon as possible after the completion of the upgrade. To run the Deferred Upgrade Tasks work queue, use the admin/bin/maintenance\_tools command:

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

### 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 PolicyCenter from 8.0.x for ContactManager

This topic lists the manual tasks required to upgrade PolicyCenter 8.0.x and ContactManager 8.0.x to PolicyCenter 8.0.3 and ContactManager 8.0.3. Before starting this upgrade process, you must have run the Guidewire upgrade and merge tools. Additionally, Guidewire recommends that you first upgrade ContactManager, integrate PolicyCenter with ContactManager, and refresh the ContactManager web APIs.

This topic includes:

- “Manually Upgrading PolicyCenter to Integrate with ContactManager” on page 91
- “File Changes in PolicyCenter Related to ContactManager” on page 92

## Manually Upgrading PolicyCenter to Integrate with ContactManager

This topic describes tasks you might have to perform to complete a PolicyCenter 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 PolicyCenter configuration. See “Upgrading the PolicyCenter 8.0.x Configuration” on page 27. Do not make changes yet to the files listed in this topic for PolicyCenter. You make those changes later as described in this topic.
2. Run the Database Upgrade tool to upgrade for the PolicyCenter database. See “Upgrading the PolicyCenter 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 PolicyCenter Studio, as described in the topics that follow.
4. Manually configure ContactManager. See “Upgrading ContactManager from 8.0.x” on page 93.

5. Integrate PolicyCenter and ContactManager as described at “Integrating ContactManager with Guidewire Core Applications” on page 49 in the *Contact Management Guide*.

## File Changes in PolicyCenter Related to ContactManager

### Web Service Version Changes

PolicyCenter now uses `wso1.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 PolicyCenter” on page 58 in the *Contact Management Guide*.

### PolicyCenter Can Generate and Send Unique IDs for New Contacts

PolicyCenter has changed the following files to enable it to send unique IDs to ContactManager to create a new contact. If you have customized any of these files or classes in your current configuration, you must manually merge your changes.

- `Contact.etcx` – The `Contact` entity in PolicyCenter has been extended with the field `ExternalID`, which stores the value of the unique ID that PolicyCenter generates.
- `ContactMapper` – The class `gw.contactmapper.ab800.ContactMapper` in PolicyCenter has a field mapping definition for the `EXTERNAL_UNIQUE_ID` field. See “PolicyCenter Mapping of Externally Specified Unique IDs” on page 253 in the *Contact Management Guide*.
- `PCContactLifecycle` – The class `gw.api.contact.PCContactLifecycle` has code that generates a new unique ID and assigns it to the `ExternalID` field of a new contact.

See “PolicyCenter Support for Creating External Unique IDs” on page 187 in the *Contact Management Guide*.

# Upgrading ContactManager from 8.0.x

This topic covers the manual steps needed to perform an upgrade of ContactManager 8.0.x to ContactManager 8.0.3. 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 93

## 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 94.

**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 PolicyCenter 8.0.x Configuration” on page 27.
2. Run the database upgrade tool to upgrade the ContactManager database. See “Upgrading the PolicyCenter 8.0.x Database” on page 51.
3. Manually configure files in ContactManager—the subject of this topic.
4. Upgrade PolicyCenter as described at “Upgrading PolicyCenter from 8.0.x for ContactManager” on page 91.

This topic includes:

- “Manually Configuring Changed Files” on page 94
- “BillingCenter Web Services Version Change” on page 94

## 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 248 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 65 in the *Contact Management Guide*

- “Configuring ContactManager-to-BillingCenter Authentication” on page 83 in the *Contact Management Guide*



# Upgrading from 7.0.x

This part describes how to perform an upgrade from PolicyCenter 7.0.x to 8.0.3.

If you are upgrading from PolicyCenter 8.0.x, see “Upgrading from 8.0.x” on page 25 instead.

If you are upgrading from PolicyCenter 4.0.x, see “Upgrading from 4.0.x” on page 207 instead.

This part includes the following topics:

- “Upgrading the PolicyCenter 7.0.x Configuration” on page 99
- “Upgrading the PolicyCenter 7.0.x Database” on page 143
- “Upgrading PolicyCenter from 7.0.x for ContactManager” on page 195
- “Upgrading ContactManager from 7.0.x” on page 199



# Upgrading the PolicyCenter 7.0.x Configuration

This topic describes how to upgrade the PolicyCenter configuration from version 7.0.x to 8.0.3.

If you are upgrading from an 8.0.x version, see “Upgrading the PolicyCenter 8.0.x Configuration” on page 27 instead.

If you are upgrading from a 4.0.x version, see “Upgrading the PolicyCenter 4.0.x Configuration” on page 209 instead.

This topic includes:

- “Overview of ContactManager Upgrade” on page 100
- “Obtaining Configurations and Tools” on page 100
- “Creating a Configuration Backup” on page 104
- “Removing Patches” on page 104
- “Removing Language Packs” on page 104
- “Updating Infrastructure” on page 104
- “Launching the PolicyCenter 8.0.3 Configuration Upgrade Tool” on page 105
- “Configuration Upgrade Tool Automated Steps” on page 106
- “Using the PolicyCenter 8.0.3 Upgrade Tool Interface” on page 114
- “Merging Product Model Files” on page 120
- “Configuration Merging Guidelines” on page 121
- “Data Model Merging Guidelines” on page 122
- “Updating Product Model API Calls” on page 126
- “Merging PolicyCenter Typelists” on page 127
- “Changes to the Logging API” on page 129
- “Merging Enhancements” on page 132

- “Updating PolicyPeriodPlugin.gs” on page 133
- “Consider Enabling Check for Small Cost Changes” on page 133
- “Merging Claim Details PCF Files” on page 134
- “Adding DDL Configuration Options to database-config.xml” on page 134
- “Merging Changes to Field Validators” on page 134
- “Renaming PCF files According to Their Modes” on page 135
- “Merging Display Properties” on page 135
- “Merging Other Files” on page 135
- “Fixing Gosu Issues” on page 136
- “Upgrading Rules to PolicyCenter 8.0.3” on page 138
- “Translating New Display Properties and Typecodes” on page 140
- “Validating the PolicyCenter 8.0.3 Configuration” on page 140
- “Importing Policy Forms” on page 141
- “Building and Deploying PolicyCenter 8.0.3” on page 141

## Overview of ContactManager Upgrade

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

- For information on upgrading ContactManager, see “Upgrading ContactManager from 7.0.x” on page 199.
- For information on upgrading PolicyCenter files used to integrate with ContactManager, see “Upgrading PolicyCenter from 7.0.x for ContactManager” on page 195.

## 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.

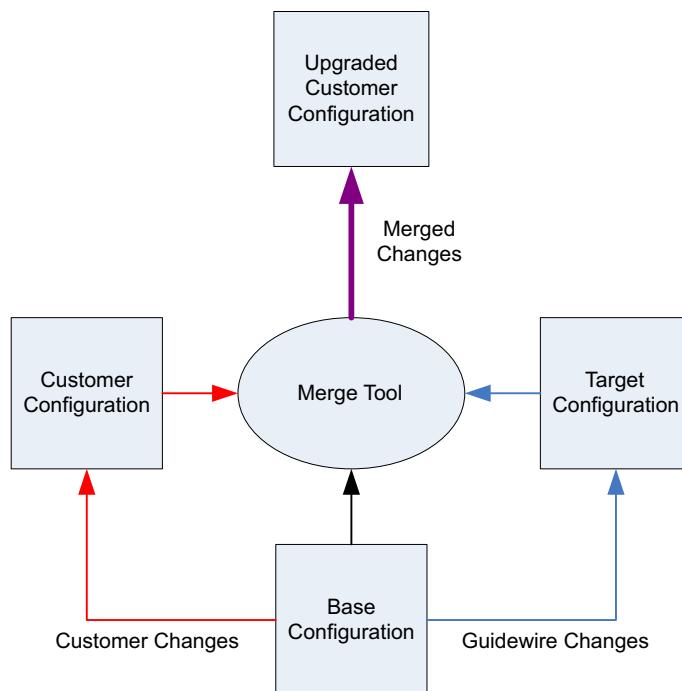
**Base** – The unedited, original configuration on which you based your customer configuration. The base configuration is included in directories within `/modules` other than `/configuration`.

**Customer** – The configuration you are now using and will upgrade. This is the base configuration of the PolicyCenter 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 PolicyCenter 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 PolicyCenter version.

**Target** – The unedited, original configuration of PolicyCenter 8.0.3 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 PolicyCenter 8.0.3 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 PolicyCenter.
5. Click Upgrade Diff Reports - PolicyCenter or Upgrade Diff Reports - ContactManager.
6. Click Upgrade From *base version*.
7. Click Upgrade To 8.0.3.

## Specifying Configuration and Tool Locations

The PolicyCenter 8.0.3 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 3 Professional. 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 PolicyCenter 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 `PolicyCenter/modules/ant/upgrade.properties` file of the target PolicyCenter 8.0.3 environment. Remove the pound sign, '#', preceding each property to uncomment the line and then specify values. Use double backslashes in paths. For example, `C:\\\\PolicyCenter`. 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 PolicyCenter directory of the current customer environment. This directory contains <code>/bin</code> and other PolicyCenter 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 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> .
<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. 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 display order, from left to right, for versions of a file viewed in the difference editor tool specified by <code>upgrader.merge.tool</code> . By default, the tool displays, left to right, the versions of a file in this order:  <code>NewFile OldFile ConfigFile</code> in which:  <code>NewFile</code> is the unmodified target version provided with PolicyCenter 8.0.3. <code>OldFile</code> is the original base version. <code>ConfigFile</code> is your configured version.  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:  <b>Araxis Merge Professional</b> <code>upgrader.merge.tool.arg.order = NewFile OldFile ConfigFile</code> <b>Beyond Compare 3 Professional</b> <code>upgrader.merge.tool.arg.order = NewFile ConfigFile OldFile</code> <b>P4Merge</b> <code>upgrader.merge.tool.arg.order = OldFile NewFile ConfigFile</code> You might need to configure the display of your merge tool to show three panels.
<code>upgrader.steps.class</code>	The class to run to execute the configuration upgrade automated steps. If you are upgrading PolicyCenter 4.0 or newer, then leave this property commented out.
<code>exclude.pattern</code>	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.

## 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.

### Backing up the Configuration

Guidewire recommends that you track PolicyCenter configuration changes in a source code control system. Before upgrading, have a labeled version of your entire pre-upgrade PolicyCenter 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.

Before upgrading PolicyCenter, commit all changes in all open Product Designer change lists. Uncommitted changes are discarded during the upgrade process.

### Backing up the Product Model

Normally, backing up the existing `config` directory backs up the product model. You can back it up separately by saving a copy of the `config/resources/productmodel` directory.

## Removing Patches

If you have applied any patches from Guidewire to PolicyCenter, remove the patches before you run the configuration upgrade. Patches are specific to the pre-upgrade version. If you 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 PolicyCenter. See “Upgrading Display Languages” on page 28 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 PolicyCenter 8.0.3 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 101.

Also see the *PolicyCenter 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 PolicyCenter 8.0. See “Template Page PCF Files Removed” on page 46 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, PolicyCenter 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 85 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 PolicyCenter 8.0 format.

As of PolicyCenter 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 PolicyCenter. 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 PolicyCenter. 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 8.0.3 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 PolicyCenter 8.0.3 to a PolicyCenter 8.0.3 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 PolicyCenter 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 PolicyCenter 8.0.3 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter/modules/p1/xsd/metadata/datamodel.xsd and <http://guidewire.com/typelists> to PolicyCenter/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 PolicyCenter 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 50 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.

## Adding Default Currency on CovTermOpt and CovTermPack Nodes

The upgrade adds the default currency to `CovTermOpt` and `CovTermPack` nodes in product model lookup files. The default currency is used for single-currency mode.

## Adding Currency Filters to Choice Lookup Table Configurations

The upgrade adds currency `<Filter>` nodes to `<CovTermOptLookup>` and `<CovTermPackLookup>` nodes in `lookuptables.xml`.

## Adding CovTermLimits to DirectCovTermPattern

The upgrade adds a `<CovTermLimits>` node to `<DirectCovTermPattern>` nodes and moves any `DefaultValue`, `MinVal`, and `MaxVal` properties to the `<CovTermLimits>` node.

## Adding CovTermDefault to OptionCovTermPattern

The upgrade adds a `<CovTermDefault>` child to the `<OptionCovTermPattern>` node and moves the `defaultValue` property from the `<OptionCovTermPattern>` to the `<CovTermDefault>` child node.

## Adding Default Currency to PolicyLinePattern

The upgrade adds the system default currency as an `<AvailableCoverageCurrency>` for a `<PolicyLinePattern>`.

## Setting Default Answer for Questions with BooleanCheckbox Format

For Questions with a questionFormat of BooleanCheckbox with no defaultAnswer, the upgrade sets defaultAnswer to FALSE.

## Setting questionPostOnChange to auto

The upgrade sets questionPostOnChange to auto if no value is specified for questionPostOnChange already.

## Normalizing Dates in the Product Model to a Standard Format

The upgrade normalizes all dates in the product model to use a standard format of yyyy-MM-dd HH:mm:ss.SSS. In some cases, the upgrade might not be able to parse a date. If the upgrade cannot parse a particular date, it reports an error in the upgrade log. If you receive this error, correct the date in the product model file by setting the date to the standard format of yyyy-MM-dd HH:mm:ss.SSS.

## Removing splitOnAnniversary from Product Line Configuration

PolicyCenter 8.0 no longer supports the splitOnAnniversary attribute in product line configuration. The upgrade removes the splitOnAnniversary attribute from any product line configurations.

# Using the PolicyCenter 8.0.3 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:

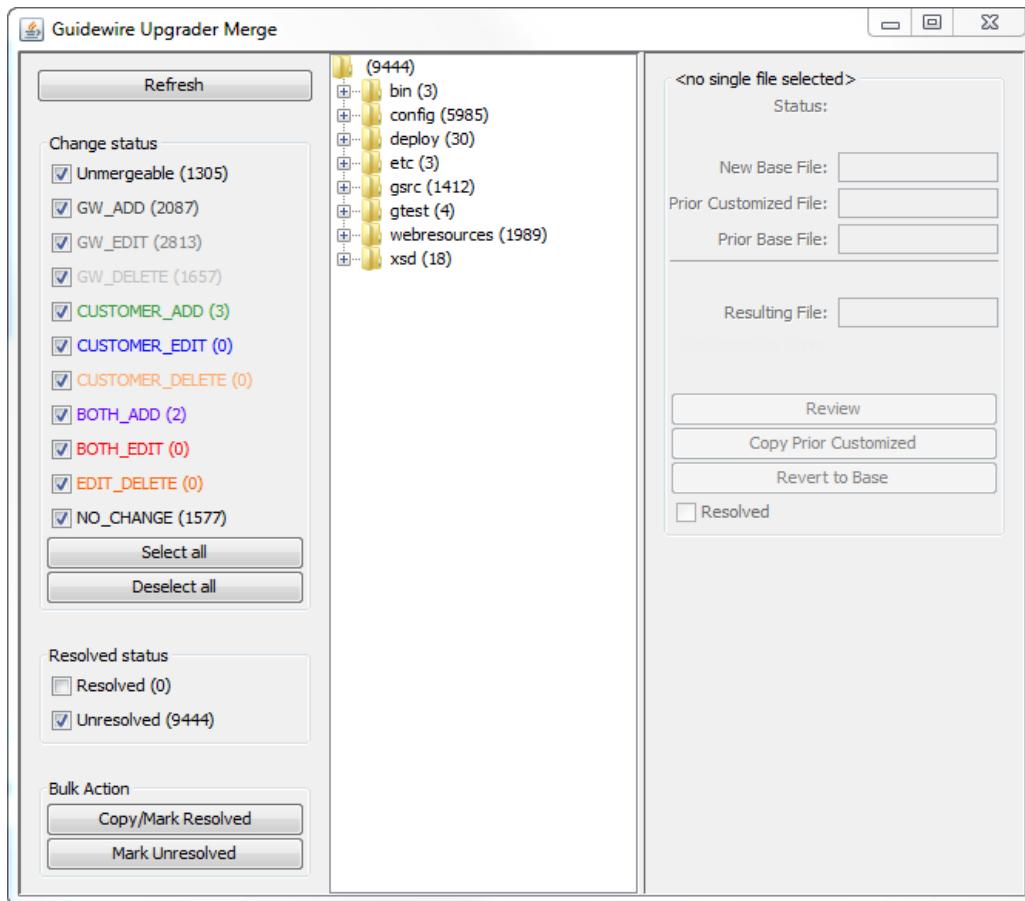
- The *customer* file.
- The *base* file, from which you configured the customer file.
- The *target* file, from PolicyCenter 8.0.3.

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 116.
- 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 and product model 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.  Typically, you use exclude.pattern to specify source control metadata files. Samples are provided in upgrade.properties for CVS and SVN.
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 <b>Delete</b> , the Configuration Upgrade Tool removes the file from the target configuration. If you click <b>Revert to Base</b> , 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 <b>Copy prior customized</b> to move the file to the target configuration.</p> <p>The <b>EDIT_DELETE</b> 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 PolicyCenter. 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>PolicyCenter 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 <b>CUSTOMER_EDIT</b> 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 PolicyCenter 8.0.3.

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 116.
- **New base file** – The new version of this file supplied by Guidewire with PolicyCenter 8.0.3. 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 PolicyCenter 8.0.3 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.

## Merging Product Model Files

Do not follow the normal upgrade process when working with product model files. The base product model content is just a starting point for custom configurations. Any modifications that Guidewire makes to the base content will already have been made to your product model if it was relevant to your business. Therefore, do not attempt to merge in content changes to the base product model files into your versions of the files. The only changes to take are syntactic changes, for which there are configuration upgrade triggers. Consequently, product model files are labelled `Unmergeable` by the Configuration Upgrade Tool.

**The normal process to merge configuration files is:**

1. Open the Configuration Upgrade Tool.
2. Select a file.
3. Merge versions of the file.
4. Save the merged file in the default location. The merged file must have the same name as the original file being resolved.
5. Close the merge tool.
6. Answer Yes to the copy question when the Configuration Upgrade Tool prompts you.

However, product model files, once configured, represent a carrier's insurance policies. Do not merge product model files with updated content delivered by Guidewire. Only update your configuration with automated syntax changes to the product model files. These syntax changes are made by automated steps of the Configuration Upgrade Tool.

In addition to the following procedure, many other files that must be merged have dependencies upon the product model of the carrier. Only merge changes by Guidewire to these files if they do not conflict with the existing product model. Some examples of such files are:

- config/lookuptables/lookuptables.xml
- Policyline-specific Gosu enhancements
- Rating Gosu classes
- Policyline-specific validation Gosu classes

**Instead, the product model-specific process is:**

1. Open the Configuration Upgrade Tool. Doing this executes automated syntax upgrades to product model files.
2. Close the Configuration Upgrade Tool.
3. Delete all of the PolicyCenter 8.0 default product model files by deleting the config/resources/productmodel folder.
4. Manually copy all files in the temporary upgrade config/resources/productmodel directory to your new configuration directory.
5. Manually copy config/locale/\*/productmodel.display.properties for each defined locale from the temporary upgrade directory to your new configuration module. If you used unmodified product model patterns in PolicyCenter 4.0, then add the contents of productmodel.display.properties from the PolicyCenter 4.0 pc module to the productmodel.display.properties in the new configuration module.

## Configuration Merging Guidelines

The first milestone of an upgrade project is to generate the Java and SOAP APIs (by running gwpc regen-java-api and gwpc 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 145.

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 PolicyCenter 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 PolicyCenter.

### 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 101.

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, PolicyCenter 8.0.3. 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, PolicyCenter enforces restrictions on the use of shared typekeys among subtypes.

### Same Field Name and Typelist with Different Column

In PolicyCenter 7.0 and earlier, if a shared typekey had the same field name and typelist, and specified a different column name, PolicyCenter created only one of the typekey columns. The shared typekeys were stored in the single column. As of PolicyCenter 8.0.3, if a shared typekey with the same field name and typelist specifies a different column name, PolicyCenter 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 PolicyCenter 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 PolicyCenter 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

PolicyCenter 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

PolicyCenter 8.0.3 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 PolicyCenter 8.0.3, 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 PolicyCenter 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

PolicyCenter 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 PolicyCenter 8.0.3. PolicyCenter 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 PolicyCenter 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 PolicyCenter.

### To generate the data dictionary

1. From the command line, navigate to the `bin` directory of the target version.
2. Run the command `gwpc 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 PolicyCenter 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

PolicyCenter 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.

If you change the extension definition to use `MonetaryAmount` rather than `Money`, the upgrade will correctly convert your entity extension data if you are upgrading from a single-currency instance. If you already manage multiple currencies, either explicitly or implicitly, such as through an extension on the account, write an `AfterUpgradeVersionTrigger` to correct currency values. The upgrade trigger must populate a currency column with a typecode value from the `Currency` typelist for each `MonetaryAmount`. Review “Upgrading Currency” on page 185 for changes made by the database upgrade. Contact Guidewire Support for assistance.

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.
- 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 where name and columnName differ:**

```
<column  
    name=" Total"  
    columnName="totalColumn"  
    type="money"/>
```

**New Total:**

```
<monetaryamount  
    name="Total"  
    amountColumnName="totalColumn"  
    soapNullOk="true" />
```

## Updating Product Model API Calls

In PolicyCenter 8.0.1 and newer, Guidewire updated several product model API classes to implement a public interface rather than extend a public abstract class. This change simplifies the API and prevents potentially dangerous methods from being exposed. The public interfaces do not include a `getByCode` method that was available on the abstract classes. Instead, a related lookup class provides the method to retrieve the product model object. All of the new interfaces are within the `gw.api.productmodel` package.

For upgrades from versions prior to 8.0.1, update your code to change calls to the `getByCode` method to the new method available on the lookup class. The new method is provided in the following table.

Old method	New method
<code>ChoiceCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
ChoiceCovTermPattern extended the public abstract class <code>CovTermPatternInternal</code> .	ChoiceCovTermPattern extends the public interface <code>CovTermPattern</code> .
<code>ClausePattern.getByCode(code)</code>	<code>ClausePatternLookup.getByCode(code)</code>
<code>ConditionPattern.getByCode(code)</code>	<code>ClausePatternLookup.getConditionPatternByCode(code)</code>
ConditionPattern extended the public abstract class <code>ClausePattern</code> .	ConditionPattern extends the public interface <code>ClausePattern</code> .
<code>CoveragePattern.getByCode(code)</code>	<code>ClausePatternLookup.getCoveragePatternByCode(code)</code>
CoveragePattern extended the public abstract class <code>ClausePattern</code> .	CoveragePattern extends the public interface <code>ClausePattern</code> .
<code>CovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
<code>CoverageCategory.getByCode(code)</code>	<code>CoverageCategoryLookup.getByCode(code)</code>
<code>CoverageSymbolPattern.getByCode(code)</code>	<code>CoverageSymbolPatternLookup.getByCode(code)</code>
<code>DirectCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
DirectCovTermPattern extended the public abstract class <code>CovTermPatternInternal</code> .	DirectCovTermPattern extends the public interface <code>CovTermPattern</code> .
<code>ExclusionPattern.getByCode(code)</code>	<code>ClausePatternLookup.getExclusionPatternByCode(code)</code>
ExclusionPattern extended the public abstract class <code>ClausePattern</code> .	ExclusionPattern extends the public interface <code>ClausePattern</code> .
<code>ModifierPatternBase.getByCode(code)</code>	<code>ModifierPatternBaseLookup.getByCode(code)</code>
<code>Offering.getByCode(code)</code>	<code>OfferingLookup.getByCode(code)</code>
<code>OptionCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
OptionCovTermPattern extended the public abstract class ChoiceCovTermPattern, which in turn extended the public abstract class <code>CovTermPatternInternal</code> .	OptionCovTermPattern extends the public interface ChoiceCovTermPattern, which in turn extends the public interface <code>CovTermPattern</code> .
<code>PackageCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
PackageCovTermPattern extended the public abstract class ChoiceCovTermPattern, which in turn extended the public abstract class <code>CovTermPatternInternal</code> .	PackageCovTermPattern extends the public interface ChoiceCovTermPattern, which in turn extends the public interface <code>CovTermPattern</code> .
<code>PolicyLinePattern.getByCode(code)</code>	<code>PolicyLinePatternLookup.getByCode(code)</code>
<code>Product.getByCode(code)</code>	<code>ProductLookup.getByCode(code)</code>
<code>Question.getByCode(code)</code>	<code>QuestionLookup.getByCode(code)</code>
<code>QuestionChoice.getByCode(code)</code>	<code>QuestionChoiceLookup.getByCode(code)</code>
<code>QuestionSet.getByCode(code)</code>	<code>QuestionSetLookup.getByCode(code)</code>
<code>RateFactorPatternBase.getByCode(code)</code>	<code>RateFactorPatternBaseLookup.getByCode(code)</code>
<code>TypekeyCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
TypekeyCovTermPattern extended the public abstract class <code>CovTermPatternInternal</code> .	TypekeyCovTermPattern extends the public interface <code>CovTermPattern</code> .

See the *Upgrade Diffs* report for more API changes.

## Merging PolicyCenter Typelists

This topic includes information specific to certain PolicyCenter typelists.

## GLCoverageFormType

In PolicyCenter versions prior to 7.0.5, the GLCoverageFormType typelist was defined in config/metadata/GLCoverageFormType.tti and its typecodes Occurrence and ClaimsMade were defined in config/extensions/GLCoverageFormType.ttx. The metadata typekey referring to the GLCoverageFormType typelist has a default value of Occurrence, which was invalid because the Occurrence typecode was defined as an extension. In PolicyCenter 7.0.5, Guidewire moved the definitions for the Occurrence and ClaimsMade typecodes into config/metadata/GLCoverageFormType.tti and removed the config/extensions/GLCoverageFormType.ttx file.

If you had added custom GLCoverageFormType typecodes in a PolicyCenter version prior to 7.0.5, move these custom typecode definitions from config/metadata/GLCoverageFormType.tti to config/extensions/GLCoverageFormType.ttx. In GLCoverageFormType.ttx, use the typelistextension tag instead of the typelist tag.

If you had deleted any GLCoverageFormType typecodes by removing the typecode definitions from GLCoverageFormType.ttx, retire those typecodes in PolicyCenter 8.0.3 rather than deleting them. Then, edit typekeys to the GLCoverageFormType typecode if necessary so that the default typecode to use is not one that is retired.

For example, the GeneralLiabilityLine policy line includes a typekey to GLCoverageFormType, defined as follows:

```
<typekey  
    default="Occurrence"  
    desc="Form of coverage (e.g. Occurrence, Claims Made)"  
    name="GLCoverageForm"  
    typelist="GLCoverageFormType"/>
```

If you had previously deleted the Occurrence typecode in a version prior to PolicyCenter 7.0.5, instead retire the typecode in PolicyCenter 8.0.3.

1. Start Studio.
2. In the Project window, navigate to configuration → config → metadata → typelist.
3. Double-click GLCoverageFormType.tti.
4. Select the Occurrence typecode.
5. Change the value of Retired to true.

Then, change the default value for the GLCoverageFormType typekey on the GeneralLiabilityLine entity.

1. In the Project window, navigate to configuration → config → metadata → entity.
2. Right-click GeneralLiabilityLine.eti and select New → Entity Extension.
3. Click OK on the dialog box. Studio opens a tab for GeneralLiabilityLine.etx.
4. Right-click the GLCoverageForm typelist and click Override.
5. Change the default from Occurrence to ClaimsMade.

In this example the typekey override changes the default to the ClaimsMade typecode instead of the retired Occurrence typecode. Adjust your implementation as needed for typecodes that you want to retire and the typecodes that you want to use as defaults.

## PercentDuplicated

The `IMAccountsReceivable` entity has a typekey referring to the `PercentDuplicated` typelist. In PolicyCenter versions prior to 7.0.5, the typekey definition was in `config/metadata/IMAccountsReceivable.eti` and the `PercentDuplicated` typelist was defined entirely in `config/extensions/PercentDuplicated.tti`. In PolicyCenter 7.0.5, Guidewire has moved the definition for the `PercentDuplicated` typelist to `config/metadata/PercentDuplicated.tti`. The typecodes are defined in `config/extensions/PercentDuplicated.ttx`.

If you have added typecodes to the `PercentDuplicated` typelist, move the typecode definitions to `config/extensions/PercentDuplicated.ttx`. This file uses the `<typelistextension>` tag rather than the `<typelist>` tag.

## ReceptacleType

The `IMAccountsReceivable` entity has a typekey to the `ReceptacleType` typelist. In PolicyCenter versions prior to 7.0.5, the typekey definition was in `config/metadata/IMAccountsReceivable.eti` and the `ReceptacleType` typelist was defined entirely in `config/extensions/ReceptacleType.tti`. In PolicyCenter 7.0.5, Guidewire has moved the definition for the `ReceptacleType` typelist to `config/metadata/ReceptacleType.tti`. The typecodes are defined in `config/extensions/ReceptacleType.ttx`.

If you have added typecodes to the `ReceptacleType` typelist, move the typecode definitions to `config/extensions/ReceptacleType.ttx`. This file uses the `<typelistextension>` tag rather than the `<typelist>` tag.

## GLStateCostType

The `GLStateCost` entity has a typekey to the `GLStateCostType` typelist. In PolicyCenter versions prior to 7.0.5, the typekey definition was in `config/metadata/GLStateCost.eti` and the `GLStateCostType` typelist was defined entirely in `config/extensions/GLStateCostType.tti`. In PolicyCenter 7.0.5, Guidewire has moved the definition for the `ReceptacleType` typelist to `config/metadata/GLStateCostType.tti`. The typecodes are defined in `config/extensions/GLStateCostType.ttx`.

If you have added typecodes to the `GLStateCostType` typelist, move the typecode definitions to `config/extensions/GLStateCostType.ttx`. This file uses the `<typelistextension>` tag rather than the `<typelist>` tag.

# Changes to the Logging API

Guidewire updated the logging API between PolicyCenter 7.0.3 and 7.0.4. 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 PolicyCenter 7.0.4, 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.pl.logging.Logger</code>	<p><code>Logger</code> was converted from a class to an interface in PolicyCenter 7.0.4. In 8.0, <code>Logger</code> is deprecated. You can update code to use <code>org.slf4j.Logger</code> instead of <code>gw.pl.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"> <li>• You cannot set logging level within the application</li> <li>• You cannot add nor remove appenders within the application</li> </ul> <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.pl.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.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());
```

## Merging Enhancements

Many line of business enhancement files have been renamed for consistency. Guidewire renamed all *TypeExt.gsx* enhancements to *TypeEnhancement.gsx*. Guidewire also renamed some *LOBLineEnhancement.gsx* to use the full name of the line of business. In a few cases, Guidewire merged enhancements into existing enhancements that were already properly named.

A file that is renamed appears twice in the Configuration Upgrade Tool. If you have not modified the enhancement, the old file name is shown under the **GW\_DELETE** filter. If you have modified the enhancement, the old file name is shown under the **EDIT\_DELETE** filter. The new file name is shown under the **GW\_ADD** filter.

The following methods have been removed from the *PolicyPeriodBaseEnhancement* and *PolicyEnhancement*:

*PolicyPeriodBaseEnhancement*

- function includes(date : Date) : boolean
- property get OOSSliceDates() : Date[]
- property get O OSSlices() : PolicyPeriod[]
- function getO OSSlices(oosSliceDates\_ : Date[]) : PolicyPeriod[]
- public property get FutureBoundDatesInPeriod() : Date[]
- public property get FutureBoundDatesInRewriteSourcePeriod() : Date[]
- public property get FutureSliceDatesInPeriod() : Date[]
- private function getFutureBoundDate(p : Policy) : Date[]

*PolicyEnhancement*

- property get BoundEditEffectiveDates() : Date[]
- property get RewrittenToNewAccountSource() : Policy
- property get RewrittenToNewAccountDestination() : Policy

## Updating PolicyPeriodPlugin.gs

As of PolicyCenter 7.0.1, `PolicyPeriodPlugin.gs` copies certain non-revisioned fields into the new preemption branch from the preempted branch. Non-revisioned fields are fields that always apply to the entire policy term. `PolicyPeriodPlugin.gs` only copies these field values into the preemption branch if their values changed from the period on which they are based. `PolicyPeriodPlugin.gs` does not copy unchanged values, and the preemption branch gets the field values from the preempting branch. If you have implemented your own `PolicyPeriodPlugin`, copy the new functionality into your custom class. See the `copyNonEffDatedFieldsForPreemption` method for an example.

PolicyCenter 7.0.1 and newer copies the following non-revisioned fields:

- `BaseState`
- `BillImmediatelyPercentage`
- `BillingMethod`
- `DepositAmount`
- `DepositCollected`
- `DepositOverridePct`
- `InvoiceStreamCode`
- `NewInvoiceStream`
- `Offering`
- `OverrideBillingAllocation`
- `PaymentPlanID`
- `PaymentPlanName`
- `PeriodEnd`
- `ProducerCodeOfRecord`
- `RateAsOfDate`
- `ReportingPatternCode`
- `Segment`
- `UWCompany`
- `WaiveDepositChange`

## Consider Enabling Check for Small Cost Changes

`CostData.gs` contains a check against amounts computed using a release prior to PolicyCenter 7.0.2 or 4.0.6. The `computeAmount` method was changed in PolicyCenter 7.0.2 and 4.0.6 to bring its behavior in line with that of the prorater. The check assures that on a policy change, if the item being rated has not changed, the new prorated amounts are the same as computed amounts from the prior release. This avoids the amounts being slightly off due to rounding, for example.

This check is expensive and could potentially slow down the overall rating response time, even if external rating is used. If you do want to use this check, consider restricting the circumstances in which the check returns `true`. For example, you could check the date range to see if the cost was created by a PolicyCenter 7.0 version prior to 7.0.2 or 4.0 version prior to 4.0.6.

In PolicyCenter 8.0.3 the check is disabled by default. If you are upgrading from a release prior to 7.0.2 or 4.0.6, and you want to enable this check, use the following procedure.

### To enable the check

1. Start Studio for PolicyCenter 8.0.3.
2. Open Classes → `gw` → `rating` → `CostData`.
3. Change the following line to set the property to `true`.

```
var _roundingGuard : boolean as GuardAgainstRoundingChange = false
```

4. Click File → Save Changes.

## Merging Claim Details PCF Files

Guidewire combined the `AccountClaimDetailsCV.pcf` and `AccountClaimDetailsDV.pcf` pages with `ClaimDetailsCV.pcf` and `ClaimDetailsDV.pcf`, respectively. Guidewire changed the usage of the `AccountClaimDetails` PCF pages to use the `ClaimDetails` pages instead. The `ClaimDetails` pages now have an argument to control whether the account version is shown.

## 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 28 in the *Installation Guide*
- “Configuring PolicyCenter to Use Oracle SecureFile LOBs” on page 35 in the *Installation Guide*
- “Configuring Table Partitioning for Oracle” on page 35 in the *Installation Guide*

## Merging Changes to Field Validators

The `<ValidatorDef>` element in `fieldvalidators.xml` accepts new attributes with PolicyCenter 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
<code>validation-level</code>	<code>none</code> <code>relaxed</code> <code>strict</code>	<code>strict</code>	The validation-level is passed to the Gosu validators. The functionality for each validation level is specific to the custom validator.
<code>validation-type</code>	<code>gosu</code> <code>regex</code>	<code>regex</code>	If <code>validation-type</code> is set to <code>regex</code> , the value of the <code>&lt;ValidatorDef&gt;</code> defines a regular expression that PolicyCenter uses to validate data entered into a field that uses the field validator.  If the <code>validation-type</code> is set to <code>gosu</code> , the value of the <code>&lt;ValidatorDef&gt;</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 PolicyCenter 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.

## Merging Display Properties

The Configuration Upgrade Tool updates display properties files, such as `display.properties`, as described in “Upgrading Display Keys” on page 112 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 PolicyCenter 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 PolicyCenter 8.0.3 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 140.

In PolicyCenter 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, PolicyCenter 8.0.3 reports a warning message if you have configuration parameters defined in `config.xml` that PolicyCenter 8.0.3 does not use. PolicyCenter ignores any unused parameters. You might have old parameters in `config.xml` that PolicyCenter does not use. If PolicyCenter 8.0.3 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 PolicyCenter 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 57 in the *New and Changed Guide*
- “What’s New and Changed in 8.0 Maintenance Releases” on page 19 in the *New and Changed Guide*

### Gosu Case Sensitivity

PolicyCenter 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 PolicyCenter 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 PolicyCenter 8.0.3, 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 PolicyCenter 8.0.3, 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 PolicyCenter 8.0.3

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 PolicyCenter 8.0.3 Rules folder within `modules/configuration/config/rules`. Use Studio to update your rules. You can use the rules in the PolicyCenter 8.0.3 folder as a comparison. Compare your custom rules to the new default 8.0.3 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 8.0.3 versions to determine what types of changes Guidewire has made.

#### To compare rules between versions using the Rule Repository Report

4. 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 PolicyCenter directory.  
If you want to compare your custom rules against the PolicyCenter 8.0.3 rules, do not move the `modules/configuration/config/rules` directory from your starting version. However, do remove the `PolicyCenter<base version>` directory from `modules/configuration/config/rules/rules` of the starting version if this directory exists.
5. Open a command window.
6. Navigate to the `bin` directory of your starting version.
7. Enter the following command:  
`gwpc regen-rulereport`  
This command creates a rule repository report XML file in `build/rules`.
8. Append the starting version number to the XML file name.

9. Restore moved directories to the starting version.
10. Install files for a fresh PolicyCenter 8.0.3 version. This is a separate configuration from the target configuration that you have merged. This version will only contain the default rules provided with PolicyCenter 8.0.3.
11. Navigate to the `bin` directory of the new PolicyCenter 8.0.3 version.

12. Enter the following command:

```
gwpc 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.

13. Append the target version number to the XML file name.

14. 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 PolicyCenter 8.0.3 rules by opening the default rules in the PolicyCenter 8.0.3 directory within **configuration** → **config** → **Rule Sets**. When you have finished updating all of your custom rules, delete the PolicyCenter 8.0.3 rules directory from `modules/configuration/config/rules`.

The PolicyCenter 8.0.3 default rules are enabled because some features depend on these rules.

## Rules Required for Free Text Search

Several rules are required for the Free Text Search functionality added in PolicyCenter 8.0 to work properly.

The first rule is:

```
modules\configuration\config\rules\EventMessage\EventFired_dir\IndexingSystem.gr.
```

The remaining rules are located within:

```
modules\configuration\config\rules\EventMessage\EventFired_dir\IndexingSystem_dir.  
• Job_dir\PurgeJob.gr  
• Policy_dir\PurgePolicy.gr  
• PolicyAddress_dir\ChangeAddress.gr  
• PolicyPeriod_dir\AddPeriod.gr  
• PolicyPeriod_dir\ChangePeriod.gr  
• PolicyPeriod_dir\PreemptedPeriod.gr  
• PolicyPeriod_dir\PurgePeriod.gr  
• PolicyPeriod_dir\RemovePeriod.gr  
• Contact.gr  
• Job.gr  
• Policy.gr  
• PolicyAddress.gr  
• PolicyPeriod.gr
```

## Translating New Display Properties and Typecodes

PolicyCenter 8.0.3 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 PolicyCenter environment, skip this procedure.

### To localize new display properties and typecodes

1. Export display keys by running the following command from your PolicyCenter 8.0.3 environment

PolicyCenter/bin directory:

```
gwpc export-110ns -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 PolicyCenter 8.0.3 environment

PolicyCenter/bin directory:

```
gwpc import-110ns -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 PolicyCenter 8.0.3 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 PolicyCenter. Do not start PolicyCenter at this point. Studio can run without connecting to the application server.

#### To validate Studio resources

1. Start Guidewire Studio by running `gwpc studio` from the `PolicyCenter\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 PolicyCenter and Resolving Errors

**IMPORTANT** In the process described in this section, do not point the PolicyCenter server at a production database. The goal of this process is to test the configuration upgrade. Create an empty database account and point PolicyCenter to this account for this process. See “Configuring the Database” on page 27 in the *Installation Guide* and “Deploying PolicyCenter to the Application Server” on page 82 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.
- PolicyCenter 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.

### Issues with Missing Availability Rows

If an availability row is missing for any product model object, the server stops with an error such as:

```
Missing availability row for availability table in AccountsRecPartCov.  
[1] AccountsRecOffPremisesProperty
```

where AccountsRecPartCov is the lookup table name and AccountsRecOffPremisesProperty is the product model object.

Fix the error in Product Designer by adding an Availability row to the product model object. In the example above, the error is in the Inland Marine policy line, Accts Receivable - Off Premises Property coverage. Click the **Availability** link and add an availability row to the table.

## Importing Policy Forms

1. Start your development server by opening a command window to `PolicyCenter/bin` and running the `gwpc dev-start` command.
2. Log in to the development server as `su`.
3. Click the **Administration** main tab.
4. Click **Import/Export Data** in the left navigation area.
5. Import the file `modules/configuration/policy_forms.xml` from the temporary upgrade directory.

## Building and Deploying PolicyCenter 8.0.3

After you apply and validate an upgrade to the configuration environment, rebuild and redeploy PolicyCenter. 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 PolicyCenter to the application server. See “Deploying PolicyCenter to the Application Server” on page 82 in the *Installation Guide* of the target version for instructions.

---

**WARNING** Do not yet start PolicyCenter. Only package the application file and deploy it to the application server. Starting PolicyCenter 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 PolicyCenter. JAR files in this location survive the upgrade process.

# Upgrading the PolicyCenter 7.0.x Database

This topic provides instructions for upgrading the PolicyCenter database to PolicyCenter 8.0.3.

If you are upgrading from a 4.0.x version, see “Upgrading the PolicyCenter 4.0.x Database” on page 261 instead.

This topic includes:

- “Upgrading Administration Data for Testing” on page 144
- “Identifying Data Model Issues” on page 145
- “Verifying Batch Process and Work Queue Completion” on page 146
- “Purging Data Prior to Upgrade” on page 146
- “Validating the Database Schema” on page 147
- “Checking Database Consistency” on page 148
- “Creating a Data Distribution Report” on page 148
- “Generating Database Statistics” on page 149
- “Creating a Database Backup” on page 150
- “Updating Database Infrastructure” on page 150
- “Preparing the Database for Upgrade” on page 150
- “Setting Linguistic Search Collation” on page 151
- “Deleting CoverageSymbolGroup from Coverage” on page 152
- “Customizing the Upgrade” on page 153
- “Disabling the Scheduler” on page 165
- “Suspending Message Destinations” on page 166
- “Configuring the Database Upgrade” on page 166
- “Checking the Database Before Upgrade” on page 173
- “Specifying ValueType on Coverage Terms” on page 173

- “Dropping Custom Rating Worksheet Tables” on page 174
- “Starting the Server to Begin Automatic Database Upgrade” on page 175
- “Viewing Detailed Database Upgrade Information” on page 189
- “Dropping Unused Columns on Oracle” on page 189
- “Reloading Rating Sample Data” on page 190
- “Exporting Administration Data for Testing” on page 191
- “Upgrading Phone Numbers” on page 192
- “Final Steps After The Database Upgrade is Complete” on page 193

## Upgrading Administration Data for Testing

You might want to create an upgraded administration data set for development and testing of rules and libraries with PolicyCenter 8.0.3. 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 191. 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) PolicyCenter production instance:
  - a. Log on to PolicyCenter 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**. PolicyCenter 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 `PolicyCenter/bin` and entering the following command:  
`gwpc dev-start`
5. Import this administration data into the development environment.
  - a. Log on to PolicyCenter 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 PolicyCenter 8.0.3. 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 27 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 PolicyCenter 8.0.3 configuration to the restored database.
10. Start the PolicyCenter 8.0.3 server to upgrade the database.
11. Export the upgraded administration data:
  - a. Start the PolicyCenter 8.0.3 server by navigating to `PolicyCenter/bin` and running the following command:  
`gwpc dev-start`
  - b. Open a browser to PolicyCenter 8.0.3.
  - 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 PolicyCenter 8.0.3.

## 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 146 and finishes with “Final Steps After The Database Upgrade is Complete” on page 193.

#### 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 PolicyCenter installation at an empty schema and starting the PolicyCenter server. See “Deploying PolicyCenter to the Application Server” on page 82 in the *Installation Guide*.
2. Complete the configuration upgrade for data model files in your starting version, according to the instructions in “Upgrading the PolicyCenter 7.0.x Configuration” on page 99. 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 66 in the *Installation Guide*.
4. Start the PolicyCenter server in your upgraded development environment. The server performs the database upgrade to PolicyCenter 8.0.3. See “Starting the Server to Begin Automatic Database Upgrade” on page 175.
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 153.

## 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 PolicyCenter as the superuser.
2. Press Alt + Shift + T. PolicyCenter 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 PolicyCenter.

### 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 `pc_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 PolicyCenter, 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 pc\_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 PolicyCenter creates an activity, the activity is added to the pc\_Workflow, pc\_WorkflowLog and pc\_WorkflowWorkItem tables. Once a user completes the activity, PolicyCenter sets the workflow status to completed. The pc\_Workflow, pc\_WorkflowLog and pc\_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.

PolicyCenter 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 PolicyCenter/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 PolicyCenter/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

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

### To run consistency checks

1. Start the PolicyCenter server if it is not already running.
2. Log in to PolicyCenter 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**.

For more information about the **Consistency Checks** page, see “**Consistency Checks**” on page 147 in the *System Administration Guide*.

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 PolicyCenter **Consistency Checks** page.

## 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 PolicyCenter 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 65 in the *System Administration Guide*.

#### To create a database distribution report

1. In config.xml, set <param name="EnableInternalDebugTools" value="true"/>.
2. Start the PolicyCenter application server.
3. Log into PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter database.

You can configure SQL Server to periodically update statistics. See your database documentation and “Configuring Database Statistics” on page 42 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 44 in the *System Administration Guide*. Note that the commands for generating statistics have moved to system\_tools instead of maintenance\_tools in the upgraded PolicyCenter.

## 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 PolicyCenter 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 PolicyCenter 8.0.3. 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 PolicyCenter. The only exception is when the PolicyCenter 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 PolicyCenter to collate search results. The `strength` attribute of the `LinguisticSearchCollation` element of `GWLocale` for the default locale in `localization.xml` specifies how PolicyCenter sorts search results. You can set the `strength` to `primary` or `secondary`.

With `LinguisticSearchCollation strength` set to `primary`, PolicyCenter searches results in a case-insensitive and accent-insensitive manner. PolicyCenter 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`, PolicyCenter treats “Renée”, “Renee”, “renee” and “reneé” the same.

With `LinguisticSearchCollation strength` set to `secondary`, PolicyCenter searches results in a case-insensitive, accent-sensitive manner. PolicyCenter 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 PolicyCenter search treats “Renee” and “renee” the same but treats “Renée” and “reneé” differently. By default, PolicyCenter 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`.

PolicyCenter 7.0 introduced configurable linguistic searching for SQL Server databases. In releases prior to PolicyCenter 7.0, PolicyCenter 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”.

PolicyCenter populates denormalized values of searchable columns to support the search collation. For example, with `LinguisticSearchCollation strength` set to `primary`, PolicyCenter stores the value “Reneé”, “Renee”, “reneé” and “reneé” in a denormalized column as “reneé”. With `LinguisticSearchCollation strength` set to `secondary`, PolicyCenter stores a denormalized value of “reneé” for “Renee” or “reneé” and stores “reneé” for “Reneé” 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, PolicyCenter repopulates the denormalized columns. Previous versions of PolicyCenter 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`, PolicyCenter 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 PolicyCenter database upgrade from a prior major version repopulates the denormalized columns regardless of the `LinguisticSearchCollation strength` value. PolicyCenter must repopulate the denormalized columns for Japanese locales to have search results obey the Japanese-only rules listed previously.

## Deleting CoverageSymbolGroup from Coverage

For upgrades from PolicyCenter 7.0.2 and earlier versions, the upgrade drops the `CoverageSymbolGroup` column from `pc_Coverage`. In PolicyCenter 7.0.3 and newer, the `CoverageSymbolGroup` foreign key has been removed from the `Coverage` entity and replaced with an enhancement property. The upgrade first checks that entities that implement the `Coverage` delegate have no data in their `CoverageSymbolGroup` foreign keys. Before upgrading, `CoverageSymbolGroup` must be `null`. This only affects Commercial Auto (formerly Business Auto in PolicyCenter 4.0) and Personal Auto coverages.

Before starting the server to begin the upgrade, delete any values in `pc_Coverage.CoverageSymbolGroup`. If you want to preserve this data, create an extension and move the data from `CoverageSymbolGroup` to the extension.

Additionally, it is possible, though very unlikely, to have configured entity extensions to have a similar issue to the one described above with `Coverage`. If that has occurred, an error will be reported during the upgrade process and the data must be fixed before continuing.

## 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** PolicyCenter 4.0 included a similar plugin interface, `IDatabaseUpgrade`. If you previously implemented `IDatabaseUpgrade` for an upgrade to PolicyCenter 4.0, 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 `gwpc regen-gosudoc` command from the PolicyCenter `bin` directory. Then, open `PolicyCenter/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.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 157.
6. Implement the `IDatamodelUpgrade` plugin with the new class.
  - a. Start Guidewire Studio 8.0.3 by entering `gwpc studio` from the `PolicyCenter/bin` directory.
  - b. In Studio, expand **configuration** → **config** → **Plugins**.
  - c. Right-click **registry** and click **New → Plugin**.
  - d. In the **Plugin** dialog, enter a name, such as `DatamodelUpgradePlugin`.
  - 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 157](#)
- “[“AfterUpgradeVersionTrigger Structure” on page 158](#)
- “[“Altering Columns to Match Data Model” on page 158](#)
- “[“Altering a Non-nullable Column to Nullable” on page 159](#)
- “[“Creating Columns” on page 159](#)
- “[“Dropping Columns” on page 160](#)
- “[“Renaming Columns” on page 160](#)
- “[“Setting a Column Value for a Specific Subtype” on page 161](#)
- “[“Creating Tables” on page 161](#)
- “[“Renaming Tables” on page 162](#)
- “[“Deleting Rows” on page 162](#)
- “[“Inserting Rows” on page 162](#)
- “[“Inserting Data Selected from Another Table” on page 162](#)
- “[“Updating Rows” on page 163](#)

### 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))  
  
ub.execute()
```

## Upgrading Archived Entities

If you implement archiving, and you make custom data model changes, then you can upgrade the archived XML using the IDatamodelUpgrade plugin.

Simple data model changes do not require a custom trigger. These include:

- Adding a new entity
- Updating denormalization columns
- Adding editable columns such as updatetime
- Adding new columns
- Changing the nullability of a column

More complex transformations or those that could result in loss of data require a version trigger. These include:

- changing a datatype (other than just length)
- migrating data from one table or column to another
- dropping a column
- dropping a table
- renaming a column
- renaming a table

PolicyCenter upgrades an archived entity as the entity is restored.

The `IDataModelChange` interface includes a `getArchivedDocumentUpgradeVersionTrigger` method that returns an `ArchivedDocumentUpgradeVersionTrigger`.

You can define custom `ArchivedDocumentUpgradeVersionTrigger` entities to modify archived XML. The `ArchivedDocumentUpgradeVersionTrigger` is an abstract class that you can extend to create your custom triggers.

Define the constructor of your custom `ArchivedDocumentUpgradeVersionTrigger` to call the constructor of the superclass and pass it a numeric value. For example:

```
construct() {  
    super(171)  
}
```

This numeric value is the extension version to which your trigger applies. If you run the upgrade against a database with a lower extension version, then your custom trigger is called. The current extension version is defined in `modules/configuration/config/extensions.properties`.

Provide an override definition of the `get Description` property to return a `String` that describes the actions of your trigger.

Provide an override definition for the `execute` function to define the actions that you want your custom trigger to make on archived XML.

When the upgrade executes your custom trigger, it wraps each XML entity in an `IArchivedEntity` object. Each typekey is wrapped in an `IArchivedTypekey` object. The upgrade operates on a single XML document at a time.

`ArchivedDocumentUpgradeVersionTrigger` provides the following key operations:

- `getArchivedEntitySet(entityName : String)` – returns an `IArchivedEntitySet` object that contains all `IArchivedEntity` objects of the given type in the XML document.

`IArchivedEntitySet` provides the following key methods:

- `rename(newEntityName : String)` – renames all rows in the set to the new name.
- `delete()` – deletes all rows in the set.
- `search(predicate : Predicate<IArchivedEntity>)` – returns a `List` of `IArchivedEntity` objects that match the given predicate.
- `create(referenceInfo : String, properties : List<Pair<String, Object>>)` – returns a new `IArchivedEntity` with the given properties.

`IArchivedEntity` provides the following key methods:

- `delete()` – deletes just this row.
- `getPropertyValue(propertyName : String)` – returns the value of the property of the given name. If the property value is a reference to another entity, this method returns an `IArchivedEntity`.
- `move(newEntityName : String)` – moves this to a new entity type of the given name. The type is created if it did not exist. You must add any required properties to the type.
- `updatePropertyValue(propertyName : String, newValue : String)` – updates the property of the given name to the given value.
- `getArchivedTypekeySet(typekeyName : String)` – returns an `IArchivedTypekeySet` object that contains all `IArchivedTypekey` objects in the given typelist.
- `getArchivedTypekey(typelistName : String, code : String)` – Returns an `IArchivedTypekey` representing the typekey.

`IArchivedTypekey` provides the following key method:

- `delete()` – deletes the typekey from the XML.

More methods are available for `IArchivedEntitySet`, `IArchivedEntity` and `IArchivedTypekey`. See the Gosu documentation for a full listing. Generate the Gosu documentation by navigating to the PolicyCenter bin directory and entering the following command:

```
gwpc regen-gosudoc
```

## Incremental Upgrade

When PolicyCenter archives an entity, it records the current data model version on the entity. PolicyCenter upgrades an archived entity as the entity is restored. The upgrader executes the necessary archive upgrade triggers incrementally on each archived XML entity, according to the data model version of the archived entity.

An entity archived at one version might not be restored and upgraded until several intermediate data model upgrades have been performed. Therefore, do not delete your custom upgrade triggers.

Consider the following situation. Note that this example is for demonstration purposes only. The version numbers included do not represent actual PolicyCenter versions but are included to explain the incremental upgrade process. Each '+' after a version number indicates a custom data model change.

### Guidewire data model changes

v6.0.0 – Entity does not have column X.

v6.0.1 – Guidewire adds column X to the entity with a default value of 100.

v6.0.2 – Guidewire updates the default value of column X to 200.

### Implementation #1 upgrade path

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.0+++ – More custom data model changes.

v6.0.2+ – column X introduced with a default value of 200.

Result of upgrade of an entity archived at v6.0.0+: column X has value 200.

In this situation, version 6.0.1 was skipped in the upgrade path. Therefore, column X is added with the default value of 200 that it has in version 6.0.2.

### Implementation #2 upgrade path

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.1+ – column X introduced with a default value of 100.

v6.0.2+ – column X already exists.

Result of upgrade of an entity archived at v6.0.0+: column X has value 100.

In this situation, column X is added in version 6.0.1 with the default value of 100. During the upgrade from version 6.0.1 to version 6.0.2, column X already exists. Therefore, the upgrader does not add the column. A custom trigger would be required to update the value of X from 100 to 200 during the upgrade from version 6.0.1 to 6.0.2.

## 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 PolicyCenter 8.0.3 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 PolicyCenter from sending messages until you have verified a successful database upgrade.

**To suspend message destinations**

1. Start the PolicyCenter server for the pre-upgrade version.
2. Log in to PolicyCenter 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.

## Configuring the Database Upgrade

You can set parameters for the database upgrade in the PolicyCenter 8.0.3 `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 PolicyCenter to the Application Server” on page 82 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 PolicyCenter encryption is applied on one or more attributes, the PolicyCenter 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, PolicyCenter:

- 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 169.
<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 PolicyCenter 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 PolicyCenter retrieves the current state of the counters at the end of the execution of the version trigger. PolicyCenter 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 and indexes on the `ArchivePartition` column on all entities that PolicyCenter can archive. 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, PolicyCenter runs the `DeferredUpgradeTasks` work queue as soon as the upgrade completes and the server is completely started. The `DeferredUpgradeTasks` work queue creates the nonessential performance indexes and indexes on archived entities. The database user must have permission to create indexes until after the `DeferredUpgradeTasks` work queue is complete.

Deferring nonessential index creation can shorten the duration of the upgrade process. The PolicyCenter 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` work queue 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` work queue has run to completion, PolicyCenter 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.

Do not use the archiving feature until the `DeferredUpgradeTasks` work queue has completed successfully.

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

If you do not opt to defer creation of nonessential indexes, PolicyCenter 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 PolicyCenter database upgrade marks columns as unused.

To configure the PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter does not generate statistics. This setting enables the upgrade to update statistics on changed objects. It also configures PolicyCenter 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, PolicyCenter 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 PolicyCenter 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 44 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 `pc_someTable_EXT`, specify a `<tablestatistics>` element for that table with the `action` attribute set to `keep`:

```
<database>
  ...
  <databasestatistics>
    <tablestatistics name="pc_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. 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 tempbldb 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 189.

## 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 PolicyCenter and you can also add custom version checks.

You can configure PolicyCenter 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 PolicyCenter 8.0.3 by running the following command from the bin directory:

```
gwpc 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, PolicyCenter 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.

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

If PolicyCenter 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, PolicyCenter runs all version checks regardless of a failure in one of the checks. During a regular upgrade, PolicyCenter 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.

## Specifying ValueType on Coverage Terms

In your pre-upgrade configuration, specify a **ValueType** on all customized direct and option coverage terms in your configuration. If **ValueType** is not specified for a custom direct or option coverage term, the upgrade will fail during the configuration upgrade with the following error:

```
[java] java.lang.IllegalStateException: ValueType needs to be set on one or more cov terms
```

**ValueType** does not need to be specified for unmodified direct or option coverage terms that are provided with the default configuration.

**WARNING** In PolicyCenter versions prior to 8.0.1, **ValueType** is an immutable field on direct and option coverage term types. The server will not start if you change the **ValueType** after you have set it on a production mode server. You can change the **ValueType** in Studio to satisfy the requirement that all customized direct and option coverage terms have a **ValueType** set. However, you cannot start the pre-upgrade server after setting a **ValueType**. As of PolicyCenter 8.0.1, **ValueType** is no longer an immutable field on direct and option coverage term types.

Check all direct and option coverage terms in your custom configuration for null **ValueType** fields. For any null **ValueType** fields in your custom configuration, specify a value type.

**To specify ValueType on custom direct and option coverage terms in PolicyCenter 4.0**

1. Open Studio in your pre-upgrade configuration.
2. In the Resources pane, expand **configuration** → **Product Model** → **Policy Lines**.
3. Select a policy line.
4. Click the **Coverages** tab.
5. Click each direct and option coverage term that is customized in your configuration, nested one level under the coverage to which it applies.
6. On the Basics tab, if the **Value Type** is **<none selected>**, select a **Value Type** from the drop-down list.
7. Save your changes.
8. Repeat this process for each custom direct and option coverage term in every policy line until all direct and option coverage terms have a value type specified.

**To specify ValueType on custom direct and option coverage terms in PolicyCenter 7.0**

1. Open Studio in your pre-upgrade configuration.
2. In the Resources pane, expand **configuration** → **Product Model** → **Policy Lines**.
3. On the Basics and Coverages tab, open each customized direct and option coverage term.
4. Click each direct and option coverage term that is customized in your configuration, nested one level under the coverage to which it applies.
5. If the **Value Type** is **<none selected>**, select a **Value Type** from the drop-down list.
6. Save your changes.
7. Repeat this process for each custom direct and option coverage term in every policy line until all direct and option coverage terms have a value type specified.

## Dropping Custom Rating Worksheet Tables

The database upgrade drops tables and entities that store rating worksheets directly. As of PolicyCenter 8.0.1, rating worksheets are stored on a **WorksheetContainer** stored on a policy. The **RatingWorksheet** delegate has been removed.

If you added custom tables and entities that implement the **RatingWorksheet** delegate, you can extend the upgrade to also drop these tables and entities.

If these custom tables and entities were in production before PolicyCenter 7.0.8, first extract any worksheet data that you wish to retain.

**To drop custom rating worksheet tables during upgrade**

1. In Studio, navigate to **configuration** → **config** → **dbupgrade** and open **emerald-dbupgrade-config.properties**.
2. For each custom entity that implements the **RatingWorksheet** delegate, add the entity name and corresponding table name to **RatingWorksheet.TableAndEntity**. Use the format **tableName:entityName** and separate entries with a comma.

## 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 PolicyCenter 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 PolicyCenter 8.0.3. 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.

---

Guidewire requires that you use a separate mirror database for reporting. If you did not do so, then you can experience problems during a database upgrade that are severe enough to prevent the upgrade.

In particular, the Premium Accounting extension package SQL scripts create special tables in the reporting database that reporting uses for storing premium accounting accrual data. The reporting scripts use these tables, but the PolicyCenter application does not. If you created these tables in a production database, then any attempt to upgrade that production database will fail.

If you encounter this situation, move data from all `pcrt_` prefixed tables to another schema. Then, drop all `pcrt_` prefixed tables from the database that you want to upgrade before starting the server to launch the upgrade.

## 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 *PolicyCenter 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 PolicyCenter 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:

- `pc_ArchiveAdminKey`
- `pc_ArchiveGraphRecord`
- `pc_ArchiveTransitionRec`
- `pc_ArchiveTypeKey`

The database archiving feature was not available for PolicyCenter.

### Removing InetSoft Reporting Support

The upgrade removes the following database elements that were involved in supporting InetSoft reporting:

- `pc_reportgroup`
- `pc_rolerptprivilege`
- `pc_rptgroup rpt`
- `pc_sreereport`

The upgrade also drops the `pc_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 PolicyCenter does not resend notifications.

### Setting Parameter for Data Files Imported

The upgrade sets the `data_files_imported` parameter to `finished` in the `pc_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 PolicyCenter.

### [Setting IndividualStacks column on WorkQueueProfilerConfig to Non-nullable](#)

The upgrade sets the `IndividualStacks` column on `pc_WorkQueueProfilerConfig` to non-nullable.

### [Changing Contact Foreign Keys on ContactAutoSyncWorkItem](#)

This is a Guidewire platform-level upgrade step. Because PolicyCenter does not use the Contact Auto Sync work queue, this step does not affect PolicyCenter.

The upgrade first checks that the `pc_ContactAutoSyncWorkItem` table is empty. Then the upgrade drops the `minContactID` and `maxContactID` foreign keys to `pc_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 `pc_Activity` to `PK_Activity`. On SQL Server, the upgrade renames the primary key index on `pc_Activity` to `pc_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:

- `pc_ArrayDataDist`
- `pc_ArraySizeCntDD`
- `pc_AssignableForKeyDataDist`
- `pc_AssignableForKeySizeCntDD`
- `pc_BeanVersionDataDist`
- `pc_BlobColDataDist`
- `pc_BooleanColDataDist`
- `pc_ClobColDataDist`
- `pc_DateAnalysisDataDist`
- `pc_DateBinnedDDDateBin`

- pc\_DateBinnedDDValue
- pc\_ForKeyDataDist
- pc\_GenericGroupCountDataDist
- pc\_HourAnalysisDataDist
- pc\_LoadInsertSelect
- pc\_LoadOperation
- pc\_LoadRowCount
- pc\_NullableColumnDataDist
- pc\_TableDataDist
- pc\_TableUpdateStats
- pc\_TypecodeCountDataDist
- pc\_TypekeyDataDist

### [Adding CPBlanketAutoNumber to Commercial Property Policy Lines](#)

For all Commercial Property policy lines, the upgrade creates a new `AutoNumberSequence` instance per policy term and updates `CPBlanketAutoNumberSeq` to point to the `AutoNumberSequence`. `PolicyLine` rows that correspond to the same policy term point to the same sequence.

This step applies to upgrades from versions prior to 7.0.1.

### [Dropping AccountAccount Staging Table](#)

The upgrade drops the `pcst_AccountAccount` staging table and removes `pc_AccountAccount.LoadCommandID`. The `AccountAccount` entity was introduced in PolicyCenter 7.0.0.

This step applies to upgrades from versions prior to 7.0.1.

### [Dropping FormPatternCovTermValue Name and Description Columns](#)

The upgrade drops the `Name` and `Description` columns from `FormPatternCovTermValue`. PolicyCenter derives coverage term value names and descriptions directly from the product model, so names and descriptions do not need to be stored with the form patterns.

This step applies to upgrades from versions prior to 7.0.1.

### [Dropping RIRiskCombination](#)

The upgrade drops the `pc_RIRiskCombination` table. This table only existed in PolicyCenter 7.0.0 and is not being used in PolicyCenter 7.0.1.

This step applies to upgrades from versions prior to 7.0.1.

### [Moving PolicyAddress from PolicyPeriod to EffectiveDatedFields](#)

The upgrade moves the `PolicyAddress` value for each `PolicyPeriod` record to the corresponding `EffectiveDatedFields` entity.

This step applies to upgrades from versions prior to 7.0.1.

### [Resynchronizing Product Model Lookups](#)

The upgrade deletes checksums for product model lookups from the database, causing PolicyCenter to resynchronize product model files when the server starts. This is to resolve an issue with determining the appropriate availability lookup row in certain situations.

This step applies to upgrades from versions prior to 7.0.1.

### Dropping ArchiveID from PolicyPeriod

The upgrade drops the `pc_PolicyPeriod.ArchiveID` column. This column was not used by PolicyCenter.

This step applies to upgrades from versions prior to 7.0.1.

### Dropping FormPatternCovTerm Name and Description Columns

The upgrade drops the `Name` and `Description` columns from `FormPatternCovTerm`. PolicyCenter derives coverage term names and descriptions directly from the product model, so names and descriptions do not need to be stored with the form patterns.

This step applies to upgrades from versions prior to 7.0.1.

### Moving RIRiskNumber and Dropping RIRiskVLContainer

The upgrade copies the risk number from the `RIRiskVLContainer` to each associated `RIRiskVersionList` and deletes the `pc_RIRiskVLContainer` table.

This step applies to upgrades from versions prior to 7.0.1.

### Adding Denormalized Primary Address Columns to Contact

The upgrade adds denormalized primary address columns on `pc_Contact` and populates these columns with values from `pc_Address`. This is done to improve performance of account searches when minimal name and any of the minimal primary address fields are specified. This step adds and populates the following columns on `pc_Contact`:

- City
- State
- PostalCode
- Country
- CityDenorm
- PostalCodeDenorm

PolicyCenter keeps these columns synchronized with their corresponding values on `pc_Address`. PolicyCenter also includes a database consistency check that checks that the `pc_Contact` and `pc_Address` values are synchronized.

This step applies to upgrades from versions prior to 7.0.2.

### Dropping Extractable and OverlapTable from AutoNumberSequence

The upgrade drops `Extractable` and `OverlapTable` delegate columns from `AutoNumberSequence`. The upgrade drops the `admin` and `archivepartition` columns from `pc_AutoNumberSequence`.

This step applies to upgrades from versions prior to 7.0.3.

### Setting Referenced Column on AccountContactRole

The upgrade sets the `Referenced` Boolean column on `pc_AccountContactRole` to `true` if there is any locked policy period with a `PolicyContactRole` referring to the `AccountContactRole`.

This step applies to upgrades from versions prior to 7.0.3.

## Setting Referenced Column on AccountLocation

The upgrade sets the Referenced Boolean column on pc\_AccountLocation to true if there is any locked policy period with a PolicyLocation referring to the AccountLocation.

This step applies to upgrades from versions prior to 7.0.3.

## Setting Referenced Column on Address

The upgrade sets the Referenced Boolean column on pc\_Address to true if there is any locked policy period with a PolicyAddress referring to the Address.

This step applies to upgrades from versions prior to 7.0.3.

## Dropping CoverageSymbolGroup from Coverage

The upgrade drops the CoverageSymbolGroup column from pc\_Coverage. The upgrade first checks that entities that implement the Coverage delegate have no data in their CoverageSymbolGroup foreign keys.

The CoverageSymbolGroup foreign key has been removed from the Coverage entity and replaced with an enhancement property. Before upgrading, CoverageSymbolGroup must be nulled out. This only affects Business Auto coverages.

Additionally, it is possible, though very unlikely, to have configured entity extensions to have a similar issue to the one described above with Coverage. If that has occurred, an error will be reported during the upgrade process and the data will need to be fixed before continuing.

This step applies to upgrades from versions prior to 7.0.3.

## Dropping City and PostalCode from Contact

The upgrade drops unnecessary PrimaryAddress denormalized columns City and PostalCode from pc\_Contact.

This step applies to upgrades from versions prior to 7.0.3.

## Populating PNIContactDenorm on PolicyPeriod

The upgrade adds a PNIContactDenorm column to pc\_PolicyPeriod. The upgrade then populates pc\_PolicyPeriod.PNIContactDenorm with the primary named insured's contact on the policy.

This step applies to upgrades from versions prior to 7.0.3.

## Replacing Index on CalcRoutineDefinition

The upgrade drops the index on Code, Version, and Retired on pc\_CalcRoutineDefinition. A new index is defined in CalcRoutineDefinition.etcx. The new index adds the Jurisdiction column to Code, Version, and Retired. By adding the Jurisdiction column, multiple CalcRoutineDefinitions with the same Code and Version can exist.

This step applies to upgrades from versions prior to 7.0.4.

## Replacing RateTableDataType Double Typecode with Decimal

The upgrade replaces the Double typecode on pct1\_RateTableDataType with a Decimal typecode.

This step applies to upgrades from versions prior to 7.0.4.

## Rename RateBook.BookVersion to RateBook.BookEdition

The upgrade renames pc\_RateBook.BookVersion to pc\_RateBook.BookEdition.

This step applies to upgrades from versions prior to 7.0.4.

## Dropping Form-related PMLockedEntity and PMLockedFields Entries

The upgrade drops form-related PMLockedEntity and PMLockedField entries, including:

- FormPattern
- ProductForm
- ProductPattern
- PolicyLineForms
- ProductFormPattern
- ProductForms

This step applies to upgrades from versions prior to 7.0.5.

## Populating Default Code for All Rate Table Operands

The upgrade adds an ArgumentSourceSetCode column to pc\_CalcStepDefOperand and sets the value of ArgumentSourceSetCode to DEFAULT for records where TableCode is not null.

This step applies to upgrades from versions prior to 7.0.5.

## Deleting Impact Analysis Test Prep Batch Process

The upgrade deletes the Impact Analysis Test Prep batch process and the related tables pc\_ImpactAnalysisPeriod and pc\_ImpactAnalysisCase.

This step applies to upgrades from versions prior to 7.0.5.

## Dropping Description from CalcStepDefinition

The upgrade drops the pc\_CalcStepDef.Description column.

This step applies to upgrades from versions prior to 7.0.6.

## Converting CalcStepDefinition Columns from Text to MediumText

The upgrade first checks that the Notes and SectionComment columns on pc\_CalcStepDef can be truncated to fit in a mediumtext type column. If any rows have columns that are too long for mediumtext, the upgrade reports an error and provides an SQL query to find the rows. If all rows can be truncated, the upgrade converts Notes and SectionComment to mediumtext.

This step applies to upgrades from versions prior to 7.0.6.

## Populating the Business Vehicle Foreign Key on BACost

For BusinessVehicleCovCost subtypes of BACost, the upgrade populates the BusinessVehicle foreign key with the corresponding Vehicle from BusinessVehicleCov.

This step applies to upgrades from versions prior to 7.0.6.

## Setting RateTableDefinition EntityName Values where Null

The upgrade sets each null EntityName on RateTableDefinition to the value RateFactorRow.

This step applies to upgrades from versions prior to 7.0.6.

### Populating PolicyTermArchiveState on PolicyTerm

This upgrade step converts `PolicyTerm.Archived` to `PolicyTerm.PolicyTermArchiveState`. The upgrade queries the database for all `PolicyPeriod` records that have foreign keys to a `PolicyTerm`.

If all `PolicyPeriod` records for a `PolicyTerm` have a non-null `ArchiveState`, then the upgrade sets `PolicyTermArchiveState` to `TC_FULLYARCHIVED`.

If all `PolicyPeriod` records for a `PolicyTerm` have a null `ArchiveState`, then the upgrade leaves `PolicyTermArchiveState` set to `TC_NOTARCHIVED`. This is the default value of `PolicyTermArchiveState`.

If the `PolicyPeriod` records for a `PolicyTerm` have a mix of null and not null `ArchiveState` values, then the upgrade sets `PolicyTermArchiveState` to `TC_PARTIALLYARCHIVED`.

This step applies to upgrades from versions prior to 7.0.7.

### Populating Attachment Inclusion

The upgrade creates a `pc_RIAttachmentInclusion` table if the table does not yet exist. If the table exists, but does not include a `PolicyTerm` column, the upgrade creates the `PolicyTerm` column. The upgrade then segments inclusions by policy term and makes inclusions retireable.

### Regenerating InstrumentedWorker and Dropping InstrumentedWorkerTask

The database upgrade drops the `pc_InstrumentedWorker` and `pc_InstrumentedWorkerTask` tables. The `pc_InstrumentedWorker` table is regenerated when PolicyCenter starts and includes data model changes that Guidewire made to the table between versions. The `pc_InstrumentedWorkerTask` table replaces the `pc_InstrumentedWorkerTask` table used in versions prior to PolicyCenter 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:

- `BusinessWeek`
- `Region`
- `Role`
- `UWAuthorityProfile`

### Dropping ClusterInfo

The upgrade drops the `pc_ClusterInfo` table. This table is renamed `pc_BatchServer`. The new table is created following the upgrade when PolicyCenter starts. The `pc_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.

PolicyCenter 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 `pc_ClusterMemberData` table.

## Updating SpatialPoint on Address

The upgrade updates the `SpatialPoint` column on `pc_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 `pc_ProcessHistory`.

## Dropping WorkQueueName Column from WorkQueueWorkerControl

The upgrade drops the `pc_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 `pc_Activity`, `pc_Address`, and `pc_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 `pc_dbConsistCheckRun`.
- drops the `Subtype` column from `pc_dbConsistCheckQueryExec`.
- updates null values of the `TableName` and `ThreadName` columns of `pc_dbConsistCheckQueryExec` to the value `UNKNOWN`.

## Upgrading Database Statistics Tables

The upgrade makes the following changes to tables involved in gathering database statistics for PolicyCenter:

- deletes all `pc_ProcessHistory` records for the Incremental Database Statistics process.
- drops the `Subtype` column from `pc_DatabaseUpdateStats`, `pc_TableUpdateStats`, and `pc_TableUpdateStatsStatement`.
- sets null values for `pc_TableUpdateStatsStatement.ObjectName` and `pc_TableUpdateStatsStatement.UpdateStatsStatement` to `UNKNOWN`.
- drops `pc_DatabaseUpdateStats.NumThreads`.
- drops `pc_DatabaseUpdateStats.Deletes`, `pc_TableUpdateStats.Inserts` and `pc_TableUpdateStats.RowCount`.

## Dropping Upgrade-related Tables

The upgrade drops the following tables:

- `pc_UpgradeDBParameterPair`
- `pc_UpgradeDBParameterRow`
- `pc_UpgradeDBParameterSet`

## Dropping AddressBookFingerprint from Contact and ContactCategoryScore

The upgrade drops the AddressBookFingerprint column from pc\_Contact and pc\_ContactCatsScore. The upgrade also drops the AddressBookFingerprint property from the Contact and ContactCategoryScore entities.

## Populating Original Effective Date of Policy

The upgrade creates an OriginalEffectiveDate column on pc\_Policy and populates the column with the start date of the policy. The upgrade determines the start date of the policy by finding the earliest PeriodStart value in pc\_PolicyPeriod for the matching policyID with a PolicyPeriodStatus of Bound.

## Upgrading Currency

The database upgrade runs several steps that upgrade currency values.

### Upgrading Contact Table

This step upgrades the pc\_Contact table by correctly repopulating the AccountHolderCount field and setting a default currency type for PreferredSettlementCurrency.

### Populating Preferred Coverage and Settlement Currency on PolicyPeriod

The upgrade populates PreferredCoverageCurrency and PreferredSettlementCurrency on pc\_PolicyPeriod with the default Currency.

### Adding Monetary Amount Currency Fields

The upgrade creates a currency field for all MonetaryAmounts and populates this field with the default currency.

### Populating Default Currencies on Account

The upgrade populates the PreferredCoverageCurrency and PreferredSettlementCurrency columns on Account to the default currency.

### Populating Default Currency Fields on RIAGreement and RIProgram

The upgrade populates the Currency fields of RIAGreement and RIProgram with the default currency. The upgrade also populates the ReinsuranceCurrency column on Reinsurable with the default currency.

### Populating ProducerCodeCurrency with Default Currency for each ProducerCode

The upgrade creates a ProducerCodeCurrency table with ProducerCodeId and Currency columns. The upgrade then populates ProducerCodeId with the corresponding ID from the ProducerCode table and populates Currency with the default currency.

### Populating Billing Amounts for Costs and Transactions

For Cost and delegates of Cost, the upgrade adds and populates the following columns:

- ActualAmountBilling
- ActualTermAmountBilling
- OverrideAmountBilling
- OverrideTermAmountBilling
- StandardAmountBilling
- StandardTermAmountBilling
- ActualAmountBillingCurrency
- ActualTermAmountBillingCurrency

- `OverrideAmountBillingCurrency`
- `OverrideTermAmountBillingCurrency`
- `StandardAmountBillingCurrency`
- `StandardTermAmountBillingCurrency`
- `ActualAmountCurrency`
- `ActualTermAmountCurrency`
- `OverrideAmountCurrency`
- `OverrideTermAmountCurrency`
- `StandardAmountCurrency`
- `StandardTermAmountCurrency`

The upgrade populates the currency columns with the default currency in rows where the corresponding amount column is not null.

The upgrade populates the settlement billing columns with the value of the base column. For example, `ActualAmountBilling` is set to the value of `ActualAmount`.

For `Transaction` and delegates of `Transaction`, the upgrade adds and populates the following columns:

- `AmountBilling`
- `AmountBillingCurrency`
- `AmountCurrency`

The upgrade populates these columns in the same manner as the `Cost` columns.

#### **Populating Default Currency on Delegates**

The upgrade adds a `Currency` column and populates it with the default currency for entities that implement the following delegates:

- `Coverable`
- `Coverage`
- `Exclusion`
- `PolicyCondition`

#### [\*\*Adding and Populating LastNotifiedCancellationDate on Cancellation\*\*](#)

The upgrade sets the `LastNotifiedCancellationDate` on `Cancellation` to the `CancellationDate` of the `PolicyPeriod` referenced by `SelectedVersion` if the `InitialNotificationDate` of the `Cancellation` is not null.

#### [\*\*Splitting UserRoleAssignment Table\*\*](#)

The upgrade splits `pc_UserRoleAssign` into `AccountUserRoleAssign`, `JobUserRoleAssign`, and `PolicyUserRoleAssign` tables. The upgrade moves records from `pc_UserRoleAssign` to the corresponding new table depending on whether `AccountID`, `JobID`, or `PolicyID` is not null. For example if a record has a non-null `AccountID`, the record is moved to the `AccountUserRoleAssign` table.

#### [\*\*Dropping UserGroupStats\*\*](#)

The upgrade drops the `UserGroupStats` table.

## Renaming UserStats Batch Process to TeamScreens

The upgrade changes the TypeCode of the UserStats record in the pct1\_BatchProcessType table to TeamScreens.

## Populating DisplayText on RateTableMatchOp

The upgrade populates the DisplayText column on RateTableMatchOp with the value of the Name column.

## Changing PolicyDriver.LicenseState and OfficialID.State to Jurisdiction IDs

The upgrade modifies pc\_PolicyDriver.LicenseState and pc\_OfficialID.State values from type State to type Jurisdiction.

## Checking for Successful Loading of emerald-dbupgrade-config.properties

The upgrade attempts to load properties from emerald-dbupgrade-config.properties. If the upgrade fails to read the file or load properties, it reports an error and stops the upgrade.

## Changing PriorPolicy to Extendable

PriorPolicy is an extendable class in PolicyCenter 8.0. The upgrade adds a Subtype column to pc\_PriorPolicy and sets Subtype for all records to PriorPolicy.

## Renaming Deferred Upgrade Batch Process

The upgrade renames the DeferredUpgrade batch process type to DeferredUpgradeTasks.

## Truncating pc\_Dynamic\_Assign

The upgrade truncates the pc\_Dynamic\_Assign table.

## Dropping pc\_t1\_Template

The upgrade drops the pc\_t1\_Template table.

## Dropping Columns from WorkItem Tables

The upgrade drops the AvailableSince and LastUpdateTime columns from all pc\_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.

## Dropping Contact.CityKanjiDenorm

The upgrade drops the pc\_Contact.CityKanjiDenorm column if it exists.

## Creating AgencyBillPlan Records

For each record in pc\_Organization with a non-null AgencyBillPlanID, the upgrade creates a record in pc\_AgencyBillPlan with the organization ID and AgencyBillPlanID.

## Dropping Rating Worksheet Tables

The upgrade drops tables and entities that store `RatingWorksheets` directly. As of PolicyCenter 8.0.1, rating worksheets are stored on a `WorksheetContainer` stored on a policy. The `RatingWorksheet` delegate has been removed.

## Moving CommissionPlanID from ProducerCode to ProducerCodeCurrency

The upgrade copies the `pc_ProducerCode.CommissionPlanID` to `pc_ProducerCodeCurrency.CommissionPlanID` and then deletes `pc_ProducerCode.CommissionPlanID`.

## Deleting Checksums for Product Model Lookups

The upgrade deletes checksums for product model lookups from the database. This forces PolicyCenter to resynchronize the database with product model files.

## Moving PolicyPeriod.NewInvoiceStream.Selected to PolicyPeriod.CustomBilling

The upgrade moves and renames the `Selected` column from `PolicyPeriod.NewInvoiceStream.Selected` to `PolicyPeriod.CustomBilling`. The `Selected` column did not indicate that the user selected to send a new invoice stream to BillingCenter. Instead, the user indicated custom billing and PolicyCenter either sends an invoice stream or modifies an existing one. PolicyCenter did not send any invoice stream information otherwise. Therefore the column has been renamed to `PolicyPeriod.CustomBilling` to better reflect its purpose.

## Creating the SelectedPaymentPlan PaymentPlanSummary

In PolicyCenter 8.0.2, the relationship between `PolicyPeriod` and `PaymentPlanSummary` was changed to a one-to-one relationship. The upgrade first checks that each non-retired `PaymentPlanSummary` record has a unique combination of `PolicyPeriod` and `BillingId`. The upgrade reports an error if it finds `PaymentPlanSummary` records with matching `PolicyPeriod` and `BillingId`. If the upgrade reports this error, either retire or remove the duplicate rows and restart the upgrade. When a record is restored from the archive it is upgraded to the current version. If a duplicate `PaymentPlanSummary` is detected during restoration, PolicyCenter marks the duplicate `PaymentPlanSummary` as retired.

If the upgrade finds no errors, it creates a new `SelectedPaymentPlan` `PaymentPlanSummary` for all `PolicyPeriods` and removes obsolete `PaymentPlanSummary` types.

## Deleting Checksums for Product Model Lookups and Lookup Rows

PolicyCenter 8.0.3 has a restructured product model. The upgrade deletes checksums for product model lookups and lookup rows from the database to allow resynchronization of product model files when the server starts.

The upgrade deletes all rows from the following tables:

- `pc_CondLookup`
- `pc_CovLookup`
- `pc_CovTermLookup`
- `pc_CovTermOptLookup`
- `pc_CovTermPackLookup`
- `pc_ExclLookup`
- `pc_ModifierLookup`
- `pc_OfferingLookup`
- `pc_ProductLookup`
- `pc_ProductModifierLookup`
- `pc_ProdRateFactorLookup`

- pc\_RatingFactorLookup
- pc\_QuestionLookup
- pc\_QuestionSetLookup

## Viewing Detailed Database Upgrade Information

PolicyCenter 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 PolicyCenter server if it is not already running.
2. Log in to PolicyCenter 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 PolicyCenter 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. PolicyCenter 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
```

## Reloading Rating Sample Data

If you are using any rating data provided by Guidewire, such as calcRoutines, rateBooks, rateTableDefinition, parameter sets, and so forth, remove all existing rating data, and reload new rating data.

**To reload rating sample data**

1. Start the PolicyCenter server.

2. Remove the old rating sample data by running the following Gosu script against the database:

```

uses gw.transaction.Transaction
uses gw.api.database.Query

function findEntity<T extends KeyableBean>() : List<T>{
    var q = Query.make(T)
    q.startsWith("PublicID", "pc:", false /*ignoreCase*/)
    return q.select().toList()
}

Transaction.runWithNewBundle(\ bundle -> {
    findEntity<RateBook>().each(\ rb -> bundle.add(rb).remove())
    findEntity<RateTableDefinition>().each(\ rt -> bundle.add(rt).remove())
    findEntity<RateTableMatchOpDefinition>().each(\ rb -> bundle.add(rb).remove())
    findEntity<RateFactorRow>().each(\ rb -> bundle.add(rb).remove())
    findEntity<CoverageRateFactor>().each(\ rb -> bundle.add(rb).remove())
    findEntity<CalcRoutineDefinition>().each(\ rb -> bundle.add(rb).remove())
    findEntity<CalcRoutineParameterSet>().each(\ rb -> bundle.add(rb).remove())
}, "su")

```

3. Run the following Gosu script to reload the PolicyCenter 8.0.3 rating sample data:

```

uses gw.transaction.Transaction
uses gw.sampledata.small.SmallSampleRatingData
uses gw.sampledata.tiny.TinySampleRatingData

Transaction.runWithNewBundle(\ bundle -> {
    var tinyData = new TinySampleRatingData()
    tinyData.load()
    var sampleData = new SmallSampleRatingData()
    sampleData.load()
}, "su")

```

## 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 PolicyCenter 8.0.3. 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 144. 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 PolicyCenter 8.0.3 server by navigating to `PolicyCenter/bin` and running the following command:  
`gwpc dev-start`
  - b. Open a browser to PolicyCenter 8.0.3.
  - 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 PolicyCenter 8.0.3. 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 27 in the *Installation Guide* for instructions to configure the database account.
3. Install a new PolicyCenter 8.0.3 development environment. Connect this development environment to the new database account that you created in step 2. See the *PolicyCenter 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 PolicyCenter 8.0.3 development server by navigating to `PolicyCenter/bin` and running the following command:  
`gwpc dev-start`
  - b. Open a browser to PolicyCenter 8.0.3.
  - 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

PolicyCenter 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.

PolicyCenter 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 PolicyCenter or restored from the archive.

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 PolicyCenter 7.0 and PolicyCenter 8.0 phone numbers. The `parsePhoneNumber` method parses a string into a `GWPhoneNumber` object if possible. `GWPhoneNumber` is an interface that defines a standard PolicyCenter 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 PolicyCenter 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 PolicyCenter. Phone numbers in PolicyCenter may become out of sync with ContactManager while the ContactManager Phone Number Normalizer work queue is running. It is safe to sync contacts in PolicyCenter that become out of sync with ContactManager. When you run the Phone Number Normalizer work queue for PolicyCenter, 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 database phone search and 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.

Wait for the Phone Number Normalizer work queue to complete before refreshing the Solr index.

## 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 147
- “Checking Database Consistency” on page 148, including “Checking that Contacts Have Unique Addresses” on page 194
- “Creating a Data Distribution Report” on page 148
- “Generating Database Statistics” on page 149. You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded PolicyCenter in a production environment.
- “Completing Deferred Upgrade” on page 194
- “Backing up the Database After Upgrade” on page 194

## Checking that Contacts Have Unique Addresses

An Address cannot be shared by more than one Contact. PolicyCenter 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 PolicyCenter 8.0.3 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 work queue as soon as possible after the completion of the upgrade. To run the Deferred Upgrade Tasks work queue, use the admin/bin/maintenance\_tools command:

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

## 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 PolicyCenter from 7.0.x for ContactManager

This topic lists the manual tasks required to upgrade PolicyCenter 7.0.x and ContactManager 7.0.x to PolicyCenter 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 PolicyCenter with ContactManager, and refresh the ContactManager web APIs.

**See also**

- “Upgrading the PolicyCenter 7.0.x Configuration” on page 99
- “Upgrading the PolicyCenter 7.0.x Database” on page 143

This topic includes:

- “Configuration File Changes in PolicyCenter” on page 196
- “Manually Upgrading PolicyCenter to Integrate with ContactManager” on page 197

## Configuration File Changes in PolicyCenter

The following table shows the files and classes used to configure PolicyCenter to work with ContactManager 7.0.x and the files that replace them in PolicyCenter 8.0.x. You can use this table as a reference for the list of files you see in the configuration upgrade tool.

PolicyCenter 7.0	PolicyCenter 8.0
ContactSystemPlugin Name of both plugin registry, <code>ContactSystemPlugin.xml</code> , and plugin interface, <code>ContactSystemPlugin.java</code>	ContactSystemPlugin Name of both the plugin registry, <code>ContactSystemPlugin.gwp</code> , and the plugin interface, <code>ContactSystemPlugin.java</code>  See “Integrating ContactManager with PolicyCenter in QuickStart” on page 57 in the <i>Contact Management Guide</i> .
ABContactSystemPlugin Plugin class that implements <code>ContactSystemPlugin.java</code> . Package name – <code>gw.plugin.contact.ab700</code>	ABContactSystemPlugin (ab800 version) Plugin class that implements <code>ContactSystemPlugin.java</code> . Package name – <code>gw.plugin.contact.ab800</code> .  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 PolicyCenter and ContactManager.  Package name – <code>gw.plugin.addressbook.ab700</code>	ABContactSystemPlugin (ab700 version) Plugin class that implements <code>ContactSystemPlugin.java</code> . Package name – <code>gw.plugin.contact.ab700</code> .  When ContactManager 7.0 is installed with PolicyCenter 8.0, register this plugin implementation.
ContactMapper.gs Name of the class that maps contact data as XML between PolicyCenter and ContactManager.  Package name – <code>gw.contactmapper.ab800</code>	ContactMapper.gs This XML mapping class has been completely changed in PolicyCenter 8.0. It supports integration with ContactManager 8.0.  Package name – <code>gw.contactmapper.ab800</code>  See “ContactMapper Class” on page 247 in the <i>Contact Management Guide</i> .
pc-to-cm-type-mapping.xml and cm-to-pc-type-mapping.xml Name of the files used to specify how to map differing entity names and typecodes between PolicyCenter and ContactManager.	ContactIntegrationXMLMapper.gs This XML mapping class supports integration with ContactManager 7.0.  Package name – <code>gw.contactmapper.ab700</code>  PCNameMapper.gs This Gosu class replaces the two XML mapping files in PolicyCenter 8.0.  Package name – <code>gw.contactmapper.ab800</code>  See “Core Application Mapping” on page 145 in the <i>Contact Management Guide</i> .

PolicyCenter 7.0	PolicyCenter 8.0
ContactAPI.gs The web service that implements ABCClientAPI to provide a way for ContactManager to call into PolicyCenter.	ContactAPI.gs (ab800 version) Package name – gw.webservice.pc.pc800.contact
Package name – gw.webservice.pc.pc700.contact	ContactAPI.gs (ab700 version) This web service supports integration with ContactManager 7.0. Package name – gw.webservice.pc.pc700.contact

## Manually Upgrading PolicyCenter to Integrate with ContactManager

This topic describes tasks you might have to perform to complete a PolicyCenter 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 PolicyCenter configuration. See “Upgrading the PolicyCenter 7.0.x Configuration” on page 99. Do not make changes yet to the files listed in this topic for PolicyCenter. You make those changes later as described in this topic.
2. Run the Database Upgrade tool to upgrade for the PolicyCenter database. See “Upgrading the PolicyCenter 7.0.x Database” on page 143.
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 PolicyCenter Studio, as described in the steps that follow.
4. Manually configure ContactManager. See “Upgrading ContactManager from 7.0.x” on page 199.
5. Integrate PolicyCenter and ContactManager as described at “Integrating ContactManager with Guidewire Core Applications” on page 49 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 PolicyCenter 7.0.x files `pc-to-ab-data-mapping.xml` and `ab-to-pc-data-mapping.xml`. In PolicyCenter 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 `PCNameMapper` with the names of any extended subentities and any typelists whose names differ between PolicyCenter and ContactManager. In PolicyCenter 8.0.x Studio press `CTRL+N` and enter `PCNameMapper` 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 PolicyCenter” on page 196, the PolicyCenter 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 247 in the *Contact Management Guide*.

- As described in “Configuration File Changes in PolicyCenter” on page 196, 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 197 in the *Contact Management Guide*.

#### See also

- “Working with Contact Mapping Files” on page 144 in the *Contact Management Guide*
- “Core Application Mapping” on page 145 in the *Contact Management Guide*

## Parameter transactionId Removed from ContactManager Web Services

As shown in the following table, the ContactManager web services ABClientAPI 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>ABClientAPI and ABContactAPI 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"> <li>• ABClientAPI <ul style="list-style-type: none"> <li>• <code>.mergeContacts</code></li> <li>• <code>.removeContact</code></li> <li>• <code>.updateContact</code></li> </ul> </li> <li>• ABContactAPI <ul style="list-style-type: none"> <li>• <code>.createContact</code></li> <li>• <code>.removeContact</code></li> <li>• <code>.updateContact</code></li> </ul> </li> </ul>	<p>ABClientAPI and ABContactAPI 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 85 in the *Integration Guide*
- “ABClientAPI Interface” on page 244 in the *Contact Management Guide*
- “ABContactAPI Web Service” on page 237 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 199
- “Configuration File Changes in ContactManager” on page 200

## Database Upgrade Steps in ContactManager

The database upgrade for ContactManager is the same as that for PolicyCenter, except for the one manual step described in this topic. For information on preparing for database upgrade, see “Upgrading the PolicyCenter 7.0.x Database” on page 143.

### 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 PolicyCenter 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 200.

**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 PolicyCenter 7.0.x Configuration” on page 99.
2. Run the database upgrade tool to upgrade the ContactManager database. See “Upgrading the PolicyCenter 7.0.x Database” on page 143.
3. Manually configure files in ContactManager—the subject of this topic.
4. Upgrade PolicyCenter as described at “Upgrading PolicyCenter from 7.0.x for ContactManager” on page 195.

## Manually Configuring Changed Files

If you have customized any of the classes described in the table at “Configuration File Changes in ContactManager” on page 200, 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 <b>Class Search</b> dialog in non-project mode. <b>Note:</b> Press CTRL+N twice to turn on non-project class searching.  See “ABContactAPI Web Service” on page 237 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"><li>• ABCClientAPI<ul style="list-style-type: none"><li>.mergeContacts</li><li>.removeContact</li><li>.updateContact</li></ul></li><li>• ABContactAPI<ul style="list-style-type: none"><li>.createContact</li><li>.removeContact</li><li>.updateContact</li></ul></li></ul>	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)
<b>See also</b> <ul style="list-style-type: none"><li>• “Setting Guidewire Transaction IDs” on page 85 in the <i>Integration Guide</i></li><li>• “ABCClientAPI Interface” on page 244 in the <i>Contact Management Guide</i></li><li>• “ABContactAPI Web Service” on page 237 in the <i>Contact Management Guide</i></li></ul>	
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 <b>Class Search</b> dialog in non-project mode. <b>Note:</b> Press CTRL+N twice to turn on non-project class searching.  See “ABVendorEvaluationAPI Web Service” on page 241 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 247 in the <i>Contact Management Guide</i> .

ContactManager 7.0	ContactManager 8.0
	<b>ContactIntegrationXMLMapper.gs</b> This XML mapping class supports integration with version 7.0 core applications.
	Package name – <code>gw.contactmapper.ab700</code>
<b>ClientSystemPlugin</b> Name of the plugin interface: <code>ClientSystemPlugin.java</code> . Package name: <code>gw.plugin</code> .	<b>ClientSystemPlugin</b> No name change. Read-only file is now visible in Studio if you do a CTRL+N search for <code>ClientSystemPlugin</code> .
<b>ClaimSystemPlugin.xml</b> Name of plugin registry. Navigate in Resource pane to configuration → Plugins → gw → plugin → <code>ClientSystemPlugin</code>	<b>ClaimSystemPlugin.gwp</b> Name of plugin registry. Navigate in Project window to configuration → config → Plugins → Registry
<b>CCClaimSystemPlugin</b> Plugin class to register when ClaimCenter 7.0 is installed. Extends <code>AbstractClientSystemPlugin.gs</code> . Package name – <code>gw.plugin.claim.cc700</code> .	<b>CCClaimSystemPlugin (cc800 version)</b> 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.
	<b>CCClaimSystemPlugin (cc700 version)</b> 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.
<b>PolicySystemPlugin.xml</b> Name of plugin registry. Navigate in Resource pane to configuration → Plugins → gw → plugin → <code>ClientSystemPlugin</code>	<b>PolicySystemPlugin.gwp</b> Name of plugin registry. Navigate in Project window to configuration → config → Plugins → Registry
<b>PCPolicySystemPlugin</b> Plugin class that extends <code>AbstractClientSystemPlugin.gs</code> . Package name – <code>gw.plugin.policy.pc700</code> .	<b>PCPolicySystemPlugin (pc800 version)</b> 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.
	<b>PCPolicySystemPlugin (pc700 version)</b> 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.
<b>BillingSystemPlugin.xml</b> Name of plugin registry. Navigate in Resource pane to configuration → Plugins → gw → plugin → <code>ClientSystemPlugin</code>	<b>BillingSystemPlugin.gwp</b> 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 <b>Class Search</b> dialog in non-project mode. <b>Note</b> Press CTRL+N twice to turn on non-project class searching. See “ABCClientAPI Interface” on page 244 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 244 in the *Contact Management Guide*.

### Changes to Contact Mapping in ContactManager

As described in “Configuration File Changes in ContactManager” on page 200, 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 247 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 200. 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 87 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 4.0.x

This part describes how to perform an upgrade from PolicyCenter 4.0.x to 8.0.3.

If you are upgrading from PolicyCenter 7.0.x, see “Upgrading from 7.0.x” on page 97 instead.

This part includes the following topics:

- “Upgrading the PolicyCenter 4.0.x Configuration” on page 209
- “Upgrading the PolicyCenter 4.0.x Database” on page 261
- “Upgrading Integrations and Gosu from 4.0.x” on page 321



# Upgrading the PolicyCenter 4.0.x Configuration

This topic describes how to upgrade the PolicyCenter configuration from version 4.0.x.

If you are upgrading from an 8.0.x version, see “Upgrading the PolicyCenter 8.0.x Configuration” on page 27 instead.

If you are upgrading from a 7.0.x version, see “Upgrading the PolicyCenter 7.0.x Configuration” on page 99 instead.

This topic includes:

- “Obtaining Configurations” on page 210
- “Creating a Configuration Backup” on page 214
- “Removing Patches” on page 214
- “Removing Language Packs” on page 214
- “Updating Infrastructure” on page 214
- “Upgrading the PolicyCenter 4.0 Configuration to 7.0” on page 215
- “PolicyCenter 7.0 Upgrade Tool Automated Steps” on page 215
- “Configuring the PolicyCenter 8.0 Upgrade Tool” on page 218
- “Launching the PolicyCenter 8.0 Configuration Upgrade Tool” on page 220
- “PolicyCenter 8.0.3 Configuration Upgrade Tool Automated Steps” on page 220
- “Using the PolicyCenter 8.0.3 Upgrade Tool Interface” on page 229
- “Merging Product Model Files” on page 235
- “Configuration Merging Guidelines” on page 236
- “Data Model Merging Guidelines” on page 237
- “Updating Product Model API Calls” on page 241
- “Merging PolicyCenter Typelists” on page 242

- “Upgrading the Business Auto Line Configuration” on page 244
- “Changes to the Logging API” on page 246
- “Merging CADiffTree.xml and BADiffTree.xml” on page 249
- “Changes to Iterators in PCF Files” on page 249
- “Updating Namespace on Files Loaded by GX Models” on page 249
- “Merging Enhancements” on page 249
- “Updating PolicyPeriodPlugin.gs” on page 250
- “Consider Enabling Check for Small Cost Changes” on page 251
- “Merging systables.xml” on page 251
- “Merging Claim Details PCF Files” on page 251
- “Adding DDL Configuration Options to database-config.xml” on page 252
- “Merging Changes to Field Validators” on page 252
- “Renaming PCF files According to Their Modes” on page 252
- “Merging compatibility-xsd.xml” on page 253
- “Merging Display Properties” on page 253
- “Merging Other Files” on page 254
- “Fixing Gosu Issues” on page 254
- “Upgrading Rules to PolicyCenter 8.0.3” on page 256
- “Running PCF Iterator Upgrade” on page 258
- “Translating New Display Properties and Typecodes” on page 258
- “Validating the PolicyCenter 8.0.3 Configuration” on page 259
- “Importing Policy Forms” on page 260
- “Building and Deploying PolicyCenter 8.0.3” on page 260

## 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 PolicyCenter 4.0 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 PolicyCenter version that you currently run with your custom configuration applied.

**Intermediate** – A PolicyCenter 7.0 installation. You run the automated steps of the PolicyCenter 7.0 Configuration Upgrade Tool to convert your PolicyCenter 4.0 customer configuration to PolicyCenter 7.0. You do not need to perform a manual configuration merge to PolicyCenter 7.0. Then you use the PolicyCenter 8.0.3 Configuration Upgrade Tool to convert the PolicyCenter 7.0 configuration to PolicyCenter 8.0.3. You do not need to upgrade the database to 7.0 before upgrading it to 8.0.3. You can upgrade the database from 4.0 to 8.0.3 in one procedure.

**Target** – The unedited, original configuration of PolicyCenter 8.0.3 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 PolicyCenter 7.0 configuration and the target PolicyCenter 8.0.3 configuration.

Unzip the target PolicyCenter 8.0.3 and intermediate PolicyCenter 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 PolicyCenter.
5. Click Upgrade Diff Reports - PolicyCenter or Upgrade Diff Reports - ContactManager.
6. Click Upgrade From *base version*.
7. Click Upgrade To 8.0.3.

## Specifying Configuration Locations for PolicyCenter 7.0 Upgrade Tool

The PolicyCenter 7.0 Configuration Upgrade Tool requires the location of the custom PolicyCenter environment that you are upgrading. Define this path in the `PolicyCenter/modules/ant/upgrade.properties` file of the intermediate PolicyCenter 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:\\\\PolicyCenter`. 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 PolicyCenter directory of the current customer environment. This directory contains <code>/bin</code> and other PolicyCenter 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 PolicyCenter 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><b>Araxis Merge Professional</b></p>
	<pre>upgrader.merge.tool.arg.order = NewFile OldFile ConfigFile</pre>
	<p><b>Beyond Compare 3 Professional</b></p>
	<pre>upgrader.merge.tool.arg.order = NewFile ConfigFile Oldfile</pre>
	<p><b>P4Merge</b></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.

### Backing up the Configuration

Guidewire recommends that you track PolicyCenter configuration changes in a source code control system. Before upgrading, have a labeled version of your entire pre-upgrade PolicyCenter 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.

Before upgrading PolicyCenter, commit all changes in all open Product Designer change lists. Uncommitted changes are discarded during the upgrade process.

### Backing up the Product Model

Normally, backing up the existing `config` directory backs up the product model. You can back it up separately by saving a copy of the `config/resources/productmodel` directory.

## Removing Patches

If you have applied any patches from Guidewire to PolicyCenter, remove the patches before you run the configuration upgrade. Patches are specific to the pre-upgrade version. If you 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 PolicyCenter. See “Upgrading Display Languages” on page 28 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 PolicyCenter 4.0 Configuration to 7.0

Use the PolicyCenter 7.0 Configuration Upgrade Tool to upgrade the configuration from 4.0 to 7.0. You do not need to perform a merge at this step. You only need to run the automated steps of the PolicyCenter 7.0 Configuration Upgrade Tool.

### Launching the PolicyCenter 7.0 Configuration Upgrade Tool

#### To launch the PolicyCenter 7.0 Configuration Upgrade Tool

1. Open a command window.
2. Navigate to the `modules/ant` directory of the PolicyCenter 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 PolicyCenter 7.0 Configuration Upgrade Tool interface. You do not need to use the PolicyCenter 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
```

## PolicyCenter 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 PolicyCenter 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 PolicyCenter 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/locale/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 PolicyCenter 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 PolicyCenter 8.0.3 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 PolicyCenter 8.0.3 to a PolicyCenter 8.0.3 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 PolicyCenter 8.0.3” on page 256.

## 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 PolicyCenter. 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.

## Upgrading Question Sets and Questions

This step upgrades QuestionSet XML files in config/resources/productmodel/questionsets. The upgrade adds answerContainerType and lookupTableName fields to each QuestionSet and populates these fields based on the questionSetType. The upgrade also adds a lookupTableName field to each Question and populates this field based on the questionSetType of the QuestionSet that contains the Question.

QuestionSet.question- SetType	QuestionSet.answer- ContainerType	QuestionSet.lookupTableName	Question.lookupTableName
location	PolicyLocation	LocationQuestionSetLookup	LocationQuestionLookup
offering, prequal	PolicyPeriod	PeriodQuestionSetLookup	PeriodQuestionLookup
supplemental	PolicyLine	LineQuestionSetLookup	LineQuestionLookup

For more information, see “Changes to Question Sets” on page 124 in the *New and Changed Guide*.

## Converting Form Pattern XML into Import XML

PolicyCenter versions prior to 7.0 stored form patterns in product model XML files. PolicyCenter 7.0 stores form patterns in the database. This trigger converts the form patterns XML into XML files that you can import into a database. This is so that you can import the form patterns into development environments for testing before you complete the production database upgrade.

## Mapping Custom Inference Classes to Form Codes

This step creates a system table XML file, custom\_form\_inference.xml, that maps all custom inference classes to the appropriate form codes for both product form patterns and policy line form patterns. The Configuration Upgrade Tool lists custom\_form\_inference.xml under the BOTH\_EDIT filter. Resolve the file using the Configuration Upgrade Tool to get the version produced by this upgrade step.

## Deleting Form Pattern Files and Directories

The configuration upgrade deletes product form pattern files and directories and policy line form pattern files and directories from the product model.

## Deleting Form Display Keys

This step deletes product form and policy line form display keys from productmodel.display.properties for all locales.

## Creating Jurisdiction Typelist

PolicyCenter 7.0 introduces a new Jurisdiction typelist, as an addition, but not a replacement of, the State typelist. This step creates the Jurisdiction typelist extension file Jurisdiction.ttx based on the State typelist extension. The upgrade adds a category to each element in the State typelist that maps to a corresponding Jurisdiction typecode.

During the database upgrade, an upgrade trigger updates data to point to the right rows in the new `Jurisdiction_typeList` table.

See “Modifications to Typelists” on page 120 in the *New and Changed Guide*.

## Renaming System Tables

The configuration upgrade renames some system tables to be consistent with the current naming convention.

Old name	New name
<code>cancelrefund.xml</code>	<code>cancel_refund.xml</code>
<code>notificationconfigs.xml</code>	<code>notification_configs.xml</code>
<code>Short_Rate_Factors.xml</code>	<code>short_ratefactors.xml</code>
<code>taxlocations.xml</code>	<code>tax_locations.xml</code>
<code>WC_Rating_Steps.xml</code>	<code>wc_ratingsteps.xml</code>

## Configuring the PolicyCenter 8.0 Upgrade Tool

After you have run the automated steps of the PolicyCenter 7.0 Configuration Upgrade Tool, run the PolicyCenter 8.0.3 Configuration Upgrade Tool to complete the upgrade from 7.0 to 8.0.3. First, specify locations of configurations and tools used to merge the configurations. Then run the PolicyCenter 8.0.3 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 PolicyCenter 8.0.3 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. An example is Araxis Merge. Also known as a “diff tool.” Configure the merge tool to ignore end of line characters to reduce the number of potential false positives during the configuration upgrade step.

The Configuration Upgrade Tool needs the location of the PolicyCenter environment that you will upgrade. The tool stores all versions of files to be merged and merge results in a `tmp/cfg-upgrade` directory that it creates within the target environment. Define paths to the configuration and tools in the `PolicyCenter/modules/ant/upgrade.properties` file of the target PolicyCenter 8.0.3 environment. Remove the pound sign, ‘#’, preceding each property to uncomment the line and then specify values. Use double backslashes in paths. For example, `C:\\\\PolicyCenter`. 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 <code>tmp/cfg-upgrade</code> directory in the intermediate PolicyCenter 7.0 environment. This directory is created by the PolicyCenter 7.0 Configuration Upgrade Tool during upgrade from 4.0 to 7.0. This directory contains <code>modules</code> and other PolicyCenter directories.
<code>upgrader.editor.tool</code>	Path to an executable editing tool.

Property	Description
<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> .
<code>upgrader.merge.tool</code>	<p>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. 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>.</p> <p>You might need to configure the display of your merge tool to show three panels.</p>
<code>upgrader.merge.tool.arg.order</code>	<p>The display order, from left to right, for versions of a file viewed in the difference editor tool specified by <code>upgrader.merge.tool</code>. By default, the tool displays, left to right, the versions of a file in this order:</p> <pre>NewFile OldFile ConfigFile</pre> <p>in which:</p>
	<p><code>NewFile</code> is the unmodified target version provided with PolicyCenter 8.0.3.</p> <p><code>OldFile</code> is the original base version.</p> <p><code>ConfigFile</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 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><b>Araxis Merge Professional</b></p>
	<pre>upgrader.merge.tool.arg.order = NewFile OldFile ConfigFile</pre>
	<p><b>Beyond Compare 3 Professional</b></p>
	<pre>upgrader.merge.tool.arg.order = NewFile ConfigFile OldFile</pre>
	<p><b>P4Merge</b></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. Uncomment the <code>upgrader.steps.class</code> property and set it to:</p> <pre>com.guidewire.tools.upgrade2.diamond.DiamondtoEmeraldConfigUpgraderStepList.</pre>
<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>

## Launching the PolicyCenter 8.0 Configuration Upgrade Tool

### To launch the PolicyCenter 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.

## PolicyCenter 8.0.3 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 PolicyCenter 8.0. See “Template Page PCF Files Removed” on page 46 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, PolicyCenter 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=value` (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 85 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 PolicyCenter 8.0 format.

As of PolicyCenter 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 PolicyCenter. 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 PolicyCenter. 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter 8.0.3 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 PolicyCenter 8.0.3 to a PolicyCenter 8.0.3 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 PolicyCenter 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 PolicyCenter 8.0.3 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 PolicyCenter `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 PolicyCenter 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 `PolicyCenter/modules/p1/xsd/metadata/datamodel.xsd` and `http://guidewire.com/typelists` to `PolicyCenter/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 PolicyCenter 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 50 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.

## Adding Default Currency on CovTermOpt and CovTermPack Nodes

The upgrade adds the default currency to CovTermOpt and CovTermPack nodes in product model lookup files. The default currency is used for single-currency mode.

## Adding Currency Filters to Choice Lookup Table Configurations

The upgrade adds currency <Filter> nodes to <CovTermOptLookup> and <CovTermPackLookup> nodes in `lookuptables.xml`.

## Adding CovTermLimits to DirectCovTermPattern

The upgrade adds a <CovTermLimits> node to <DirectCovTermPattern> nodes and moves any `DefaultValue`, `MinVal`, and `MaxVal` properties to the <CovTermLimits> node.

## Adding CovTermDefault to OptionCovTermPattern

The upgrade adds a <CovTermDefault> child to the <OptionCovTermPattern> node and moves the `defaultValue` property from the <OptionCovTermPattern> to the <CovTermDefault> child node.

## Adding Default Currency to PolicyLinePattern

The upgrade adds the system default currency as an <AvailableCoverageCurrency> for a <PolicyLinePattern>.

## Setting Default Answer for Questions with BooleanCheckbox Format

For Questions with a `questionFormat` of `BooleanCheckbox` with no `defaultAnswer`, the upgrade sets `defaultAnswer` to `FALSE`.

## Setting `questionPostOnChange` to `auto`

The upgrade sets `questionPostOnChange` to `auto` if no value is specified for `questionPostOnChange` already.

## Normalizing Dates in the Product Model to a Standard Format

The upgrade normalizes all dates in the product model to use a standard format of `yyyy-MM-dd HH:mm:ss.SSS`. In some cases, the upgrade might not be able to parse a date. If the upgrade cannot parse a particular date, it reports an error in the upgrade log. If you receive this error, correct the date in the product model file by setting the date to the standard format of `yyyy-MM-dd HH:mm:ss.SSS`.

## Removing `splitOnAnniversary` from Product Line Configuration

PolicyCenter 8.0 no longer supports the `splitOnAnniversary` attribute in product line configuration. The upgrade removes the `splitOnAnniversary` attribute from any product line configurations.

## Using the PolicyCenter 8.0.3 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:

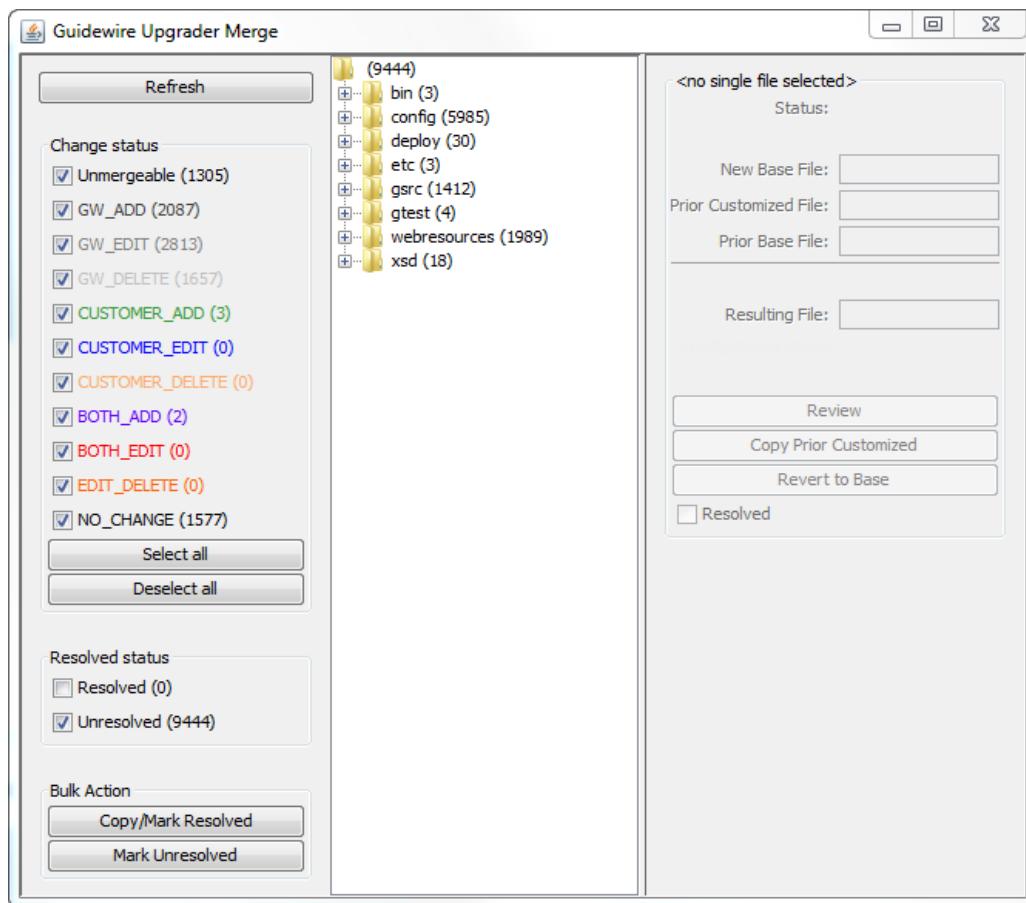
- The *customer* file.
- The *base* file, from which you configured the customer file.
- The *target* file, from PolicyCenter 8.0.3.

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 231.
- 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 and product model 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.  Typically, you use exclude.pattern to specify source control metadata files. Samples are provided in upgrade.properties for CVS and SVN.
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 <b>Delete</b> , the Configuration Upgrade Tool removes the file from the target configuration. If you click <b>Revert to Base</b> , 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 <b>Copy prior customized</b> to move the file to the target configuration.</p> <p>The <b>EDIT_DELETE</b> 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 PolicyCenter. 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>PolicyCenter 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 <b>CUSTOMER_EDIT</b> 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 PolicyCenter 8.0.3.

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 231.
- **New base file** – The new version of this file supplied by Guidewire with PolicyCenter 8.0.3. 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 PolicyCenter 8.0.3 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.

## Merging Product Model Files

Do not follow the normal upgrade process when working with product model files. The base product model content is just a starting point for custom configurations. Any modifications that Guidewire makes to the base content will already have been made to your product model if it was relevant to your business. Therefore, do not attempt to merge in content changes to the base product model files into your versions of the files. The only changes to take are syntactic changes, for which there are configuration upgrade triggers. Consequently, product model files are labelled `Unmergeable` by the Configuration Upgrade Tool.

**The normal process to merge configuration files is:**

1. Open the Configuration Upgrade Tool.
2. Select a file.
3. Merge versions of the file.
4. Save the merged file in the default location. The merged file must have the same name as the original file being resolved.
5. Close the merge tool.
6. Answer Yes to the copy question when the Configuration Upgrade Tool prompts you.

However, product model files, once configured, represent a carrier's insurance policies. Do not merge product model files with updated content delivered by Guidewire. Only update your configuration with automated syntax changes to the product model files. These syntax changes are made by automated steps of the Configuration Upgrade Tool.

In addition to the following procedure, many other files that must be merged have dependencies upon the product model of the carrier. Only merge changes by Guidewire to these files if they do not conflict with the existing product model. Some examples of such files are:

- config/lookuptables/lookuptables.xml
- Policyline-specific Gosu enhancements
- Rating Gosu classes
- Policyline-specific validation Gosu classes

**Instead, the product model-specific process is:**

1. Open the Configuration Upgrade Tool. Doing this executes automated syntax upgrades to product model files.
2. Close the Configuration Upgrade Tool.
3. Delete all of the PolicyCenter 8.0 default product model files by deleting the config/resources/productmodel folder.
4. Manually copy all files in the temporary upgrade config/resources/productmodel directory to your new configuration directory.
5. Manually copy config/locale/\*/productmodel.display.properties for each defined locale from the temporary upgrade directory to your new configuration module. If you used unmodified product model patterns in PolicyCenter 4.0, then add the contents of productmodel.display.properties from the PolicyCenter 4.0 pc module to the productmodel.display.properties in the new configuration module.

## Configuration Merging Guidelines

The first milestone of an upgrade project is to generate the Java and SOAP APIs (by running gwpc regen-java-api and gwpc 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 PolicyCenter 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 PolicyCenter.

### 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 101.

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, PolicyCenter 8.0.3. 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, PolicyCenter enforces restrictions on the use of shared typekeys among subtypes.

### Same Field Name and Typelist with Different Column

In PolicyCenter 7.0 and earlier, if a shared typekey had the same field name and typelist, and specified a different column name, PolicyCenter created only one of the typekey columns. The shared typekeys were stored in the single column. As of PolicyCenter 8.0.3, if a shared typekey with the same field name and typelist specifies a different column name, PolicyCenter 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 PolicyCenter 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 PolicyCenter 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

PolicyCenter 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

PolicyCenter 8.0.3 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 PolicyCenter 8.0.3, 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 PolicyCenter 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

PolicyCenter 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 PolicyCenter 8.0.3. PolicyCenter 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 PolicyCenter 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 PolicyCenter.

### To generate the data dictionary

1. From the command line, navigate to the `bin` directory of the target version.
2. Run the command `gwpc 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 PolicyCenter 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

PolicyCenter 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.

If you change the extension definition to use `MonetaryAmount` rather than `Money`, the upgrade will correctly convert your entity extension data if you are upgrading from a single-currency instance. If you already manage multiple currencies, either explicitly or implicitly, such as through an extension on the account, write an `AfterUpgradeVersionTrigger` to correct currency values. The upgrade trigger must populate a currency column with a typecode value from the `Currency` typelist for each `MonetaryAmount`. Review “Upgrading Currency” on page 185 for changes made by the database upgrade. Contact Guidewire Support for assistance.

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.
- 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 where name and columnName differ:**

```
<column  
    name=" Total"  
    columnName="totalColumn"  
    type="money"/>
```

**New Total:**

```
<monetaryamount  
    name="Total"  
    amountColumnName="totalColumn"  
    soapNullOk="true" />
```

## Updating Product Model API Calls

In PolicyCenter 8.0.1 and newer, Guidewire updated several product model API classes to implement a public interface rather than extend a public abstract class. This change simplifies the API and prevents potentially dangerous methods from being exposed. The public interfaces do not include a `getByCode` method that was available on the abstract classes. Instead, a related lookup class provides the method to retrieve the product model object. All of the new interfaces are within the `gw.api.productmodel` package.

For upgrades from versions prior to 8.0.1, update your code to change calls to the `getByCode` method to the new method available on the lookup class. The new method is provided in the following table.

Old method	New method
<code>ChoiceCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
ChoiceCovTermPattern extended the public abstract class <code>CovTermPatternInternal</code> .	ChoiceCovTermPattern extends the public interface <code>CovTermPattern</code> .
<code>ClausePattern.getByCode(code)</code>	<code>ClausePatternLookup.getByCode(code)</code>
<code>ConditionPattern.getByCode(code)</code>	<code>ClausePatternLookup.getConditionPatternByCode(code)</code>
ConditionPattern extended the public abstract class <code>ClausePattern</code> .	ConditionPattern extends the public interface <code>ClausePattern</code> .
<code>CoveragePattern.getByCode(code)</code>	<code>ClausePatternLookup.getCoveragePatternByCode(code)</code>
CoveragePattern extended the public abstract class <code>ClausePattern</code> .	CoveragePattern extends the public interface <code>ClausePattern</code> .
<code>CovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
<code>CoverageCategory.getByCode(code)</code>	<code>CoverageCategoryLookup.getByCode(code)</code>
<code>CoverageSymbolPattern.getByCode(code)</code>	<code>CoverageSymbolPatternLookup.getByCode(code)</code>
<code>DirectCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
DirectCovTermPattern extended the public abstract class <code>CovTermPatternInternal</code> .	DirectCovTermPattern extends the public interface <code>CovTermPattern</code> .
<code>ExclusionPattern.getByCode(code)</code>	<code>ClausePatternLookup.getExclusionPatternByCode(code)</code>
ExclusionPattern extended the public abstract class <code>ClausePattern</code> .	ExclusionPattern extends the public interface <code>ClausePattern</code> .
<code>ModifierPatternBase.getByCode(code)</code>	<code>ModifierPatternBaseLookup.getByCode(code)</code>
<code>Offering.getByCode(code)</code>	<code>OfferingLookup.getByCode(code)</code>
<code>OptionCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
OptionCovTermPattern extended the public abstract class ChoiceCovTermPattern, which in turn extended the public abstract class <code>CovTermPatternInternal</code> .	OptionCovTermPattern extends the public interface ChoiceCovTermPattern, which in turn extends the public interface <code>CovTermPattern</code> .
<code>PackageCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
PackageCovTermPattern extended the public abstract class ChoiceCovTermPattern, which in turn extended the public abstract class <code>CovTermPatternInternal</code> .	PackageCovTermPattern extends the public interface ChoiceCovTermPattern, which in turn extends the public interface <code>CovTermPattern</code> .
<code>PolicyLinePattern.getByCode(code)</code>	<code>PolicyLinePatternLookup.getByCode(code)</code>
<code>Product.getByCode(code)</code>	<code>ProductLookup.getByCode(code)</code>
<code>Question.getByCode(code)</code>	<code>QuestionLookup.getByCode(code)</code>
<code>QuestionChoice.getByCode(code)</code>	<code>QuestionChoiceLookup.getByCode(code)</code>
<code>QuestionSet.getByCode(code)</code>	<code>QuestionSetLookup.getByCode(code)</code>
<code>RateFactorPatternBase.getByCode(code)</code>	<code>RateFactorPatternBaseLookup.getByCode(code)</code>
<code>TypekeyCovTermPattern.getByCode(code)</code>	<code>CovTermPatternLookup.getByCode(code)</code>
TypekeyCovTermPattern extended the public abstract class <code>CovTermPatternInternal</code> .	TypekeyCovTermPattern extends the public interface <code>CovTermPattern</code> .

See the *Upgrade Diffs* report for more API changes.

## Merging PolicyCenter Typelists

This topic includes information specific to certain PolicyCenter typelists.

## GLCoverageFormType

In PolicyCenter versions prior to 7.0.5, the GLCoverageFormType typelist was defined in config/metadata/GLCoverageFormType.tti and its typecodes Occurrence and ClaimsMade were defined in config/extensions/GLCoverageFormType.ttx. The metadata typekey referring to the GLCoverageFormType typelist has a default value of Occurrence, which was invalid because the Occurrence typecode was defined as an extension. In PolicyCenter 7.0.5, Guidewire moved the definitions for the Occurrence and ClaimsMade typecodes into config/metadata/GLCoverageFormType.tti and removed the config/extensions/GLCoverageFormType.ttx file.

If you had added custom GLCoverageFormType typecodes in a PolicyCenter version prior to 7.0.5, move these custom typecode definitions from config/metadata/GLCoverageFormType.tti to config/extensions/GLCoverageFormType.ttx. In GLCoverageFormType.ttx, use the typelistextension tag instead of the typelist tag.

If you had deleted any GLCoverageFormType typecodes by removing the typecode definitions from GLCoverageFormType.ttx, retire those typecodes in PolicyCenter 8.0.3 rather than deleting them. Then, edit typekeys to the GLCoverageFormType typecode if necessary so that the default typecode to use is not one that is retired.

For example, the GeneralLiabilityLine policy line includes a typekey to GLCoverageFormType, defined as follows:

```
<typekey  
    default="Occurrence"  
    desc="Form of coverage (e.g. Occurrence, Claims Made)"  
    name="GLCoverageForm"  
    typelist="GLCoverageFormType"/>
```

If you had previously deleted the Occurrence typecode in a version prior to PolicyCenter 7.0.5, instead retire the typecode in PolicyCenter 8.0.3.

1. Start Studio.
2. In the Project window, navigate to configuration → config → metadata → typelist.
3. Double-click GLCoverageFormType.tti.
4. Select the Occurrence typecode.
5. Change the value of Retired to true.

Then, change the default value for the GLCoverageFormType typekey on the GeneralLiabilityLine entity.

1. In the Project window, navigate to configuration → config → metadata → entity.
2. Right-click GeneralLiabilityLine.eti and select New → Entity Extension.
3. Click OK on the dialog box. Studio opens a tab for GeneralLiabilityLine.etx.
4. Right-click the GLCoverageForm typelist and click Override.
5. Change the default from Occurrence to ClaimsMade.

In this example the typekey override changes the default to the ClaimsMade typecode instead of the retired Occurrence typecode. Adjust your implementation as needed for typecodes that you want to retire and the typecodes that you want to use as defaults.

## PercentDuplicated

The `IMAccountsReceivable` entity has a typekey referring to the `PercentDuplicated` typelist. In PolicyCenter versions prior to 7.0.5, the typekey definition was in `config/metadata/IMAccountsReceivable.eti` and the `PercentDuplicated` typelist was defined entirely in `config/extensions/PercentDuplicated.tti`. In PolicyCenter 7.0.5, Guidewire has moved the definition for the `PercentDuplicated` typelist to `config/metadata/PercentDuplicated.tti`. The typecodes are defined in `config/extensions/PercentDuplicated.ttx`.

If you have added typecodes to the `PercentDuplicated` typelist, move the typecode definitions to `config/extensions/PercentDuplicated.ttx`. This file uses the `<typelistextension>` tag rather than the `<typelist>` tag.

## ReceptacleType

The `IMAccountsReceivable` entity has a typekey to the `ReceptacleType` typelist. In PolicyCenter versions prior to 7.0.5, the typekey definition was in `config/metadata/IMAccountsReceivable.eti` and the `ReceptacleType` typelist was defined entirely in `config/extensions/ReceptacleType.tti`. In PolicyCenter 7.0.5, Guidewire has moved the definition for the `ReceptacleType` typelist to `config/metadata/ReceptacleType.tti`. The typecodes are defined in `config/extensions/ReceptacleType.ttx`.

If you have added typecodes to the `ReceptacleType` typelist, move the typecode definitions to `config/extensions/ReceptacleType.ttx`. This file uses the `<typelistextension>` tag rather than the `<typelist>` tag.

## GLStateCostType

The `GLStateCost` entity has a typekey to the `GLStateCostType` typelist. In PolicyCenter versions prior to 7.0.5, the typekey definition was in `config/metadata/GLStateCost.eti` and the `GLStateCostType` typelist was defined entirely in `config/extensions/GLStateCostType.tti`. In PolicyCenter 7.0.5, Guidewire has moved the definition for the `ReceptacleType` typelist to `config/metadata/GLStateCostType.tti`. The typecodes are defined in `config/extensions/GLStateCostType.ttx`.

If you have added typecodes to the `GLStateCostType` typelist, move the typecode definitions to `config/extensions/GLStateCostType.ttx`. This file uses the `<typelistextension>` tag rather than the `<typelist>` tag.

# Upgrading the Business Auto Line Configuration

When running the configuration upgrade tool, accept Guidewire changes to `lookuptables.xml` that change the `entityName` attribute on entries with code `BAModifier` from `ModifierLookup` to its subtype, `BAModLookup`.

Next, modify the `BusinessAutoLine-lookups.xml` file to change `ModifierLookup` to `BAModLookup`.

Finally, for rows in `pc_PolicyLine` with subtype of `BusinessAutoLine`, set the `PolicyType` to `Business Auto`. You can use a `BeforeUpgradeVersionTrigger` to set the `PolicyType`.

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 updatePolicyLinePolicyType()))
        return list
    }

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

- 5.** Create a `BeforeUpgradeVersionTrigger` Gosu class.

```
package companyName.upgrade

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

class updatePolicyLinePolicyType extends BeforeUpgradeVersionTrigger {
    construct() {
        super(57)
    }

    override function execute() {

        final var policyLineTable = getTable("pc_policyline")
        final var baTypecode = getTypeKeyID("BAPolicyType", "BA")
        final var businessAutoLineTypecode = getTypeKeyID("PolicyLine", "BusinessAutoLine")
        final var updateBuilder = policyLineTable.update()

        updateBuilder
            .set("policytype", baTypecode)
            .compare("subtype", Equals, businessAutoLineTypecode)

        updateBuilder.execute()
    }

    override property get Description() : String {
        return "set the PolicyType value on rows in pc_PolicyLine with subtype of BusinessAutoLine to Business Auto."
    }

    override function hasVersionCheck() : boolean {
        return false
    }
}
```

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

- Expand configuration → config → Plugins.
- Right-click registry and click New → Plugin.

- c. In the **Plugin** dialog, enter a name, such as `IDatamodelUpgradePlugin`.
- d. In the **Plugin** dialog, click the ... button.
- e. In the **Select Plugin Class** dialog, type `IDatamodelUpgrade` and select the `IDatamodelUpgrade` interface.
- f. In the **Plugin** dialog, click OK. Studio creates a GWP file under **Plugins → registry** with the name you entered.
- g. Click the **Add Plugin** icon (a plus sign) and select **Add Gosu Plugin**.
- h. For **Gosu Class**, enter the class you created in step 4, including the package.
- i. 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 version triggers.

## Changes to the Logging API

Guidewire updated the logging API between PolicyCenter 7.0.3 and 7.0.4. 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 PolicyCenter 7.0.4, 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 <code>xx</code> is a product-specific code such as <code>bc</code> , <code>cc</code> , or <code>pc</code> .	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.pl.logging.Logger</code>	<p>Logger was converted from a class to an interface in PolicyCenter 7.0.4. 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"> <li>• You cannot set logging level within the application</li> <li>• You cannot add nor remove appenders within the application</li> </ul> <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 gw.pl 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());
```

## Merging CADiffTree.xml and BADiffTree.xml

The Business Auto line has been renamed to Commercial Auto, and `BADiffTree.xml` has been renamed to `CADiffTree.xml`. If you have customized `BADiffTree.xml`, merge the customizations into the PolicyCenter 8.0.3 `CADiffTree.xml` file.

## Changes to Iterators in PCF Files

As of PolicyCenter 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 258.

## 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"`.

## Merging Enhancements

Many line of business enhancement files have been renamed for consistency. Guidewire renamed all `TypeExt.gsx` enhancements to `TypeEnhancement.gsx`. Guidewire also renamed some `LOBLineEnhancement.gsx` to use the full name of the line of business. In a few cases, Guidewire merged enhancements into existing enhancements that were already properly named.

A file that is renamed appears twice in the Configuration Upgrade Tool. If you have not modified the enhancement, the old file name is shown under the **GW\_DELETE** filter. If you have modified the enhancement, the old file name is shown under the **EDIT\_DELETE** filter. The new file name is shown under the **GW\_ADD** filter.

The following methods have been removed from the **PolicyPeriodBaseEnhancement** and **PolicyEnhancement**:

```
PolicyPeriodBaseEnhancement
• function includes(date : Date) : boolean
• property get O OSSSliceDates() : Date[]
• property get O OSSlices() : PolicyPeriod[]
• function getO OSSlices(oosSliceDates_ : Date[]) : PolicyPeriod[]
• public property get FutureBoundDatesInPeriod() : Date[]
• public property get FutureBoundDatesInRewriteSourcePeriod() : Date[]
• public property get FutureSliceDatesInPeriod() : Date[]
• private function getFutureBoundDate(p : Policy) : Date[]

PolicyEnhancement
• property get BoundEditEffectiveDates() : Date[]
• property get RewrittenToNewAccountSource() : Policy
• property get RewrittenToNewAccountDestination() : Policy
```

## Updating PolicyPeriodPlugin.gs

As of PolicyCenter 7.0.1, **PolicyPeriodPlugin.gs** copies certain non-revisioned fields into the new preemption branch from the preempted branch. Non-revisioned fields are fields that always apply to the entire policy term. **PolicyPeriodPlugin.gs** only copies these field values into the preemption branch if their values changed from the period on which they are based. **PolicyPeriodPlugin.gs** does not copy unchanged values, and the preemption branch gets the field values from the preempting branch. If you have implemented your own **PolicyPeriodPlugin**, copy the new functionality into your custom class. See the **copyNonEffDatedFieldsForPreemption** method for an example.

PolicyCenter 7.0.1 and newer copies the following non-revisioned fields:

- **BaseState**
- **BillImmediatelyPercentage**
- **BillingMethod**
- **DepositAmount**
- **DepositCollected**
- **DepositOverridePct**
- **InvoiceStreamCode**
- **NewInvoiceStream**
- **Offering**
- **OverrideBillingAllocation**
- **PaymentPlanID**
- **PaymentPlanName**
- **PeriodEnd**
- **ProducerCodeOfRecord**
- **RateAsOfDate**
- **ReportingPatternCode**
- **Segment**
- **UWCompany**
- **WaiveDepositChange**

## Consider Enabling Check for Small Cost Changes

`CostData.gs` contains a check against amounts computed using a release prior to PolicyCenter 7.0.2 or 4.0.6. The `computeAmount` method was changed in PolicyCenter 7.0.2 and 4.0.6 to bring its behavior in line with that of the prorater. The check assures that on a policy change, if the item being rated has not changed, the new prorated amounts are the same as computed amounts from the prior release. This avoids the amounts being slightly off due to rounding, for example.

This check is expensive and could potentially slow down the overall rating response time, even if external rating is used. If you do want to use this check, consider restricting the circumstances in which the check returns `true`. For example, you could check the date range to see if the cost was created by a PolicyCenter 7.0 version prior to 7.0.2 or 4.0 version prior to 4.0.6.

In PolicyCenter 8.0.3 the check is disabled by default. If you are upgrading from a release prior to 7.0.2 or 4.0.6, and you want to enable this check, use the following procedure.

### To enable the check

1. Start Studio for PolicyCenter 8.0.3.
2. Open Classes → `gw` → `rating` → `CostData`.
3. Change the following line to set the property to `true`.

```
var _roundingGuard : boolean as GuardAgainstRoundingChange = false
```
4. Click `File` → `Save Changes`.

## Merging systables.xml

In PolicyCenter 4.0, system table verifier classes were in the package `gw.plugin.systable.verifier`. This package was not appropriate because these classes are not related to a plugin. The system table verifier classes are specified in `systables.xml` rather than in a plugin.

In PolicyCenter 7.0, Guidewire moved the system table verifier classes to the package `gw.systables.verifier`. As a result of this move, the `config/resources/systables.xml` file has been updated.

The system table verifier classes have the annotation `@readonly` instead of `@export`. With this change, you cannot modify the default system table verifier classes. You can still view the classes in Studio. If you want a custom class, you can modify `systables.xml` to point to a custom class that you create.

For more information about configuring system tables, see “System Tables” on page 75 in the *Product Model Guide*.

## Merging Claim Details PCF Files

In PolicyCenter 7.0, Guidewire combined `AccountClaimDetailsCV.pcf` and `AccountClaimDetailsDV.pcf` with `ClaimDetailsCV.pcf` and `ClaimDetailsDV.pcf`, respectively. The usage of the `AccountClaimDetails` PCF pages has been changed to use the `ClaimDetails` pages instead. The `ClaimDetails` pages now have an argument to control whether the account version is shown.

## 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 28 in the *Installation Guide*
- “Configuring PolicyCenter to Use Oracle SecureFile LOBs” on page 35 in the *Installation Guide*
- “Configuring Table Partitioning for Oracle” on page 35 in the *Installation Guide*

## Merging Changes to Field Validators

The `<ValidatorDef>` element in `fieldvalidators.xml` accepts new attributes with PolicyCenter 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>&lt;ValidatorDef&gt;</code> defines a regular expression that PolicyCenter 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>&lt;ValidatorDef&gt;</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 PolicyCenter 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.

## Merging compatibility-xsd.xml

The PolicyCenter 7.0 Configuration Upgrade Tool generates a `compatibility-xsd.xml` file which enables existing 4.0 code referencing XSD types to continue compiling. PolicyCenter 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 PolicyCenter 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 4.0 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 4.0 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 226 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 PolicyCenter 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 PolicyCenter 8.0.3 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 258.

In PolicyCenter 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, PolicyCenter 8.0.3 reports a warning message if you have configuration parameters defined in `config.xml` that PolicyCenter 8.0.3 does not use. PolicyCenter ignores any unused parameters. You might have old parameters in `config.xml` that PolicyCenter does not use. If PolicyCenter 8.0.3 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 PolicyCenter 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 57 in the *New and Changed Guide*
- “What’s New and Changed in 8.0 Maintenance Releases” on page 19 in the *New and Changed Guide*

### Gosu Case Sensitivity

PolicyCenter 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 PolicyCenter 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 PolicyCenter 8.0.3, 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 PolicyCenter 8.0.3, 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 PolicyCenter 8.0.3

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 PolicyCenter 8.0.3 Rules folder within `modules/configuration/config/rules`. Use Studio to update your rules. You can use the rules in the PolicyCenter 8.0.3 folder as a comparison. Compare your custom rules to the new default 8.0.3 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 8.0.3 versions to determine what types of changes Guidewire has made.

#### To compare rules between versions using the Rule Repository Report

4. 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 PolicyCenter directory.

If you want to compare your custom rules against the PolicyCenter 8.0.3 rules, do not move the `modules/configuration/config/rules` directory from your starting version. However, do remove the `PolicyCenter<base version>` directory from `modules/configuration/config/rules/rules` of the starting version if this directory exists.

5. Open a command window.

6. Navigate to the `bin` directory of your starting version.

7. Enter the following command:

```
gwpc regen-rulereport
```

This command creates a rule repository report XML file in `build/rules`.

8. Append the starting version number to the XML file name.

9. Restore moved directories to the starting version.

10. Install files for a fresh PolicyCenter 8.0.3 version. This is a separate configuration from the target configuration that you have merged. This version will only contain the default rules provided with PolicyCenter 8.0.3.

11. Navigate to the `bin` directory of the new PolicyCenter 8.0.3 version.

12. Enter the following command:

```
gwpc 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.

13. Append the target version number to the XML file name.

14. 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 PolicyCenter 8.0.3 rules by opening the default rules in the PolicyCenter 8.0.3 directory within `configuration → config → Rule Sets`. When you have finished updating all of your custom rules, delete the PolicyCenter 8.0.3 rules directory from `modules/configuration/config/rules`.

The PolicyCenter 8.0.3 default rules are enabled because some features depend on these rules.

## Rules Required for Free Text Search

Several rules are required for the Free Text Search functionality added in PolicyCenter 8.0 to work properly.

The first rule is:

```
modules\configuration\config\rules\EventMessage\EventFired_dir\IndexingSystem.gr
```

The remaining rules are located within:

```
modules\configuration\config\rules\EventMessage\EventFired_dir\IndexingSystem_dir.  
• Job_dir\PurgeJob.gr  
• Policy_dir\PurgePolicy.gr  
• PolicyAddress_dir\ChangeAddress.gr  
• PolicyPeriod_dir\AddPeriod.gr  
• PolicyPeriod_dir\ChangePeriod.gr  
• PolicyPeriod_dir\PreemptedPeriod.gr  
• PolicyPeriod_dir\PurgePeriod.gr  
• PolicyPeriod_dir\RemovePeriod.gr  
• Contact.gr  
• Job.gr  
• Policy.gr  
• PolicyAddress.gr  
• PolicyPeriod.gr
```

## Running PCF Iterator Upgrade

Prior to PolicyCenter 7.0, toolbar buttons, with the exception of `CheckedValues`, did not specify an iterator. PolicyCenter used the first iterator defined in the PCF file following the button. In PolicyCenter 7.0, all iterator buttons have an `iterator` attribute set. Each toolbar button must have a specified iterator on which it operates.

PolicyCenter 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 PolicyCenter 8.0.3 directory, navigate to the `bin` directory.
2. Run the following command:

```
gwpc iterator-upgrade
```

## Translating New Display Properties and Typecodes

PolicyCenter 8.0.3 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 PolicyCenter environment, skip this procedure.

### To localize new display properties and typecodes

1. Export display keys by running the following command from your PolicyCenter 8.0.3 environment `PolicyCenter/bin` directory:  

```
gwpc 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 PolicyCenter 8.0.3 environment

PolicyCenter\bin directory:

```
gwpc import-110ns -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 PolicyCenter 8.0.3 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 PolicyCenter. Do not start PolicyCenter at this point. Studio can run without connecting to the application server.

#### To validate Studio resources

1. Start Guidewire Studio by running `gwpc studio` from the PolicyCenter\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 PolicyCenter and Resolving Errors

**IMPORTANT** In the process described in this section, do not point the PolicyCenter server at a production database. The goal of this process is to test the configuration upgrade. Create an empty database account and point PolicyCenter to this account for this process. See “Configuring the Database” on page 27 in the *Installation Guide* and “Deploying PolicyCenter to the Application Server” on page 82 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.  
PolicyCenter 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.

## Importing Policy Forms

1. Start your development server by opening a command window to `PolicyCenter/bin` and running the `gwpc dev-start` command.
2. Log in to the development server as `su`.
3. Click the **Administration** main tab.
4. Click **Import/Export Data** in the left navigation area.
5. Import the file `modules/configuration/policy_forms.xml` from the temporary upgrade directory.

## Building and Deploying PolicyCenter 8.0.3

After you apply and validate an upgrade to the configuration environment, rebuild and redeploy PolicyCenter. 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 PolicyCenter to the application server. See “Deploying PolicyCenter to the Application Server” on page 82 in the *Installation Guide* of the target version for instructions.

---

**WARNING** Do not yet start PolicyCenter. Only package the application file and deploy it to the application server. Starting PolicyCenter 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 PolicyCenter. JAR files in this location survive the upgrade process.

# Upgrading the PolicyCenter 4.0.x Database

This topic provides instructions for upgrading the PolicyCenter database to PolicyCenter 8.0.3.

If you are upgrading from a 7.0.x version, see “Upgrading the PolicyCenter 7.0.x Database” on page 143 instead.

This topic includes:

- “Upgrading Administration Data for Testing” on page 262
- “Identifying Data Model Issues” on page 263
- “Verifying Batch Process and Work Queue Completion” on page 264
- “Purging Data Prior to Upgrade” on page 264
- “Validating the Database Schema” on page 265
- “Checking Database Consistency” on page 266
- “Creating a Data Distribution Report” on page 266
- “Generating Database Statistics” on page 267
- “Creating a Database Backup” on page 268
- “Updating Database Infrastructure” on page 268
- “Preparing the Database for Upgrade” on page 268
- “Deleting CoverageSymbolGroup from Coverage” on page 269
- “Enabling Migration to 64-bit IDs (SQL Server Only)” on page 269
- “Setting Linguistic Search Collation” on page 270
- “Customizing the Upgrade” on page 271
- “Disabling the Scheduler” on page 284
- “Suspending Message Destinations” on page 285
- “Configuring the Database Upgrade” on page 285
- “Checking the Database Before Upgrade” on page 292

- “Specifying ValueType on Coverage Terms” on page 292
- “Dropping Custom Rating Worksheet Tables” on page 293
- “Starting the Server to Begin Automatic Database Upgrade” on page 294
- “Viewing Detailed Database Upgrade Information” on page 315
- “Dropping Unused Columns on Oracle” on page 316
- “Reloading Rating Sample Data” on page 317
- “Exporting Administration Data for Testing” on page 317
- “Upgrading Phone Numbers” on page 319
- “Final Steps After The Database Upgrade is Complete” on page 320

## Upgrading Administration Data for Testing

You might want to create an upgraded administration data set for development and testing of rules and libraries with PolicyCenter 8.0.3. 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 317. 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) PolicyCenter production instance:
  - a. Log on to PolicyCenter 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**. PolicyCenter 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 `PolicyCenter/bin` and entering the following command:  
`gwpc dev-start`
5. Import this administration data into the development environment.
  - a. Log on to PolicyCenter 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 PolicyCenter 8.0.3. 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 27 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 PolicyCenter 8.0.3 configuration to the restored database.
10. Start the PolicyCenter 8.0.3 server to upgrade the database.
11. Export the upgraded administration data:
  - a. Start the PolicyCenter 8.0.3 server by navigating to `PolicyCenter/bin` and running the following command:  
`gwpc dev-start`
  - b. Open a browser to PolicyCenter 8.0.3.
  - 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 PolicyCenter 8.0.3.

## 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 264 and finishes with “Final Steps After The Database Upgrade is Complete” on page 320.

**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 PolicyCenter installation at an empty schema and starting the PolicyCenter server. See “Deploying PolicyCenter to the Application Server” on page 82 in the *Installation Guide*.
2. Complete the configuration upgrade for data model files in your starting version, according to the instructions in “Upgrading the PolicyCenter 4.0.x Configuration” on page 209. 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 PolicyCenter to the Application Server” on page 82 in the *Installation Guide*.
4. Start the PolicyCenter server in your upgraded development environment. The server performs the database upgrade to PolicyCenter 8.0.3. See “Starting the Server to Begin Automatic Database Upgrade” on page 294.
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 272.

## 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 PolicyCenter as the superuser.
2. Press Alt + Shift + T. PolicyCenter 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 PolicyCenter.

### 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 `pc_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 PolicyCenter, 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 pc\_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 PolicyCenter creates an activity, the activity is added to the pc\_Workflow, pc\_WorkflowLog and pc\_WorkflowWorkItem tables. Once a user completes the activity, PolicyCenter sets the workflow status to completed. The pc\_Workflow, pc\_WorkflowLog and pc\_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.

PolicyCenter 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 PolicyCenter/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 PolicyCenter/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

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

### To run consistency checks

1. Start the PolicyCenter server if it is not already running.
2. Log in to PolicyCenter 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**.

For more information about the **Consistency Checks** page, see “**Consistency Checks**” on page 147 in the *System Administration Guide*.

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 PolicyCenter **Consistency Checks** page.

## 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 PolicyCenter 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 65 in the *System Administration Guide*.

**To create a database distribution report**

1. In config.xml, set <param name="EnableInternalDebugTools" value="true"/>.
2. Start the PolicyCenter application server.
3. Log into PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter database.

You can configure SQL Server to periodically update statistics. See your database documentation and “Configuring Database Statistics” on page 42 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 44 in the *System Administration Guide*. Note that the commands for generating statistics have moved to system\_tools instead of maintenance\_tools in the upgraded PolicyCenter.

## 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 PolicyCenter 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 PolicyCenter 8.0.3. 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.

## Deleting CoverageSymbolGroup from Coverage

For upgrades from PolicyCenter 7.0.2 and earlier versions, the upgrade drops the `CoverageSymbolGroup` column from `pc_Coverage`. In PolicyCenter 7.0.3 and newer, the `CoverageSymbolGroup` foreign key has been removed from the `Coverage` entity and replaced with an enhancement property. The upgrade first checks that entities that implement the `Coverage` delegate have no data in their `CoverageSymbolGroup` foreign keys. Before upgrading, `CoverageSymbolGroup` must be null. This only affects Commercial Auto (formerly Business Auto in PolicyCenter 4.0) and Personal Auto coverages.

Before starting the server to begin the upgrade, delete any values in `pc_Coverage.CoverageSymbolGroup`. If you want to preserve this data, create an extension and move the data from `CoverageSymbolGroup` to the extension.

Additionally, it is possible, though very unlikely, to have configured entity extensions to have a similar issue to the one described above with `Coverage`. If that has occurred, an error will be reported during the upgrade process and the data must be fixed before continuing.

## Enabling Migration to 64-bit IDs (SQL Server Only)

PolicyCenter 8.0.3 uses 64-bit `BIGINT` primary key and foreign key identifiers and `datetime2` date columns. PolicyCenter 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, PolicyCenter 8.0.3 provides a configuration parameter to enable the migration. This parameter is `MigrateToLargeIDsAndDatetime2`. By default `MigrateToLargeIDsAndDatetime2` is set to `false`.

You might want to complete the rest of the database upgrade and then perform the migration later. In that case, you do not need to do anything before the database upgrade. After the upgrade completes, use the procedure in this topic to enable the migration.

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 PolicyCenter.

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

1. Open Studio for PolicyCenter 8.0.3.
2. Open **configuration** → **config** → **config.xml**.
3. Modify the following line to set `value="true"`.  
`<param name="MigrateToLargeIDsAndDatetime2" value="false"/>`
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.  
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 290.

7. Save your changes. If you are performing the migration at the same time as the PolicyCenter 8.0.3 database upgrade, skip the rest of this procedure. If you are performing the migration separately from the PolicyCenter 8.0.3 database upgrade, continue to the next step.
8. Start the server. PolicyCenter performs the migration to 64-bit BIGINT identifiers and datetime2 date columns.

## 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 PolicyCenter. The only exception is when the PolicyCenter 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 PolicyCenter to collate search results. The `strength` attribute of the `LinguisticSearchCollation` element of `GWLocale` for the default locale in `localization.xml` specifies how PolicyCenter sorts search results. You can set the `strength` to `primary` or `secondary`.

With `LinguisticSearchCollation strength` set to `primary`, PolicyCenter searches results in a case-insensitive and accent-insensitive manner. PolicyCenter 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`, PolicyCenter treats “Renée”, “Renée”, “renee” and “renée” the same.

With `LinguisticSearchCollation strength` set to `secondary`, PolicyCenter searches results in a case-insensitive, accent-sensitive manner. PolicyCenter 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 PolicyCenter search treats “Renée” and “renee” the same but treats “Renée” and “renée” differently. By default, PolicyCenter 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`.

PolicyCenter 7.0 introduced configurable linguistic searching for SQL Server databases. In releases prior to PolicyCenter 7.0, PolicyCenter 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".

PolicyCenter populates denormalized values of searchable columns to support the search collation. For example, with `LinguisticSearchCollation strength` set to primary, PolicyCenter stores the value "Reneé", "Renee", "reneé" and "reneé" in a denormalized column as "reneé". With `LinguisticSearchCollation strength` set to secondary, PolicyCenter stores a denormalized value of "reneé" for "Renee" or "reneé" and stores "reneé" for "Reneé" 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, PolicyCenter repopulates the denormalized columns. Previous versions of PolicyCenter 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, PolicyCenter 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 PolicyCenter database upgrade from a prior major version repopulates the denormalized columns regardless of the `LinguisticSearchCollation strength` value. PolicyCenter must repopulate the denormalized columns for Japanese locales to have search results obey the Japanese-only rules listed previously.

## 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** PolicyCenter 4.0 included a similar plugin interface, `IDatabaseUpgrade`. If you previously implemented `IDatabaseUpgrade` for an upgrade to PolicyCenter 4.0, 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 `gwpc regen-gosudoc` command from the `PolicyCenter/bin` directory. Then, open `PolicyCenter/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 275.

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

- a. Start Guidewire Studio 8.0.3 by entering `gwpc studio` from the `PolicyCenter/bin` directory.
- b. In Studio, expand `configuration` → `config` → `Plugins`.
- c. Right-click `registry` and click `New` → `Plugin`.
- d. In the `Plugin` dialog, enter a name, such as `DatamodelUpgradePlugin`.
- 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 276

- “AfterUpgradeVersionTrigger Structure” on page 277
- “Altering Columns to Match Data Model” on page 277
- “Altering a Non-nullable Column to Nullable” on page 277
- “Creating Columns” on page 278
- “Dropping Columns” on page 279
- “Renaming Columns” on page 279
- “Setting a Column Value for a Specific Subtype” on page 279
- “Creating Tables” on page 280
- “Renaming Tables” on page 280
- “Deleting Rows” on page 280
- “Inserting Rows” on page 281
- “Inserting Data Selected from Another Table” on page 281
- “Updating Rows” on page 282

## 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()

```

## Upgrading Archived Entities

If you implement archiving, and you make custom data model changes, then you can upgrade the archived XML using the `IDatamodelUpgrade` plugin.

Simple data model changes do not require a custom trigger. These include:

- Adding a new entity
- Updating denormalization columns
- Adding editable columns such as `updatetime`
- Adding new columns
- Changing the nullability of a column

More complex transformations or those that could result in loss of data require a version trigger. These include:

- changing a datatype (other than just length)
- migrating data from one table or column to another
- dropping a column
- dropping a table
- renaming a column
- renaming a table

PolicyCenter upgrades an archived entity as the entity is restored.

The `IDatamodelChange` interface includes a `getArchivedDocumentUpgradeVersionTrigger` method that returns an `ArchivedDocumentUpgradeVersionTrigger`.

You can define custom `ArchivedDocumentUpgradeVersionTrigger` entities to modify archived XML. The `ArchivedDocumentUpgradeVersionTrigger` is an abstract class that you can extend to create your custom triggers.

Define the constructor of your custom `ArchivedDocumentUpgradeVersionTrigger` to call the constructor of the superclass and pass it a numeric value. For example:

```

construct() {
    super(171)
}

```

This numeric value is the extension version to which your trigger applies. If you run the upgrade against a database with a lower extension version, then your custom trigger is called. The current extension version is defined in `modules/configuration/config/extensions/extensions.properties`.

Provide an override definition of the `get Description` property to return a `String` that describes the actions of your trigger.

Provide an override definition for the `execute` function to define the actions that you want your custom trigger to make on archived XML.

When the upgrade executes your custom trigger, it wraps each XML entity in an `IArchivedEntity` object. Each typekey is wrapped in an `IArchivedTypekey` object. The upgrade operates on a single XML document at a time.

`ArchivedDocumentUpgradeVersionTrigger` provides the following key operations:

- `getArchivedEntitySet(entityName : String)` – returns an `IArchivedEntitySet` object that contains all `IArchivedEntity` objects of the given type in the XML document.

`IArchivedEntitySet` provides the following key methods:

- `rename(newEntityName : String)` – renames all rows in the set to the new name.
- `delete()` – deletes all rows in the set.
- `search(predicate : Predicate<IArchivedEntity>)` – returns a `List` of `IArchivedEntity` objects that match the given predicate.
- `create(referenceInfo : String, properties : List<Pair<String, Object>>)` – returns a new `IArchivedEntity` with the given properties.

`IArchivedEntity` provides the following key methods:

- `delete()` – deletes just this row.
- `getPropertyValue(propertyName : String)` – returns the value of the property of the given name. If the property value is a reference to another entity, this method returns an `IArchivedEntity`.
- `move(newEntityName : String)` – moves this to a new entity type of the given name. The type is created if it did not exist. You must add any required properties to the type.
- `updatePropertyValue(propertyName : String, newValue : String)` – updates the property of the given name to the given value.
- `getArchivedTypekeySet(typekeyName : String)` – returns an `IArchivedTypekeySet` object that contains all `IArchivedTypekey` objects in the given typelist.
- `getArchivedTypekey(typelistName : String, code : String)` – Returns an `IArchivedTypekey` representing the typekey.

`IArchivedTypekey` provides the following key method:

- `delete()` – deletes the typekey from the XML.

More methods are available for `IArchivedEntitySet`, `IArchivedEntity` and `IArchivedTypekey`. See the Gosu documentation for a full listing. Generate the Gosu documentation by navigating to the PolicyCenter bin directory and entering the following command:

```
gwpc regen-gosudoc
```

## Incremental Upgrade

When PolicyCenter archives an entity, it records the current data model version on the entity. PolicyCenter upgrades an archived entity as the entity is restored. The upgrader executes the necessary archive upgrade triggers incrementally on each archived XML entity, according to the data model version of the archived entity.

An entity archived at one version might not be restored and upgraded until several intermediate data model upgrades have been performed. Therefore, do not delete your custom upgrade triggers.

Consider the following situation. Note that this example is for demonstration purposes only. The version numbers included do not represent actual PolicyCenter versions but are included to explain the incremental upgrade process. Each '+' after a version number indicates a custom data model change.

#### **Guidewire data model changes**

v6.0.0 – Entity does not have column X.

v6.0.1 – Guidewire adds column X to the entity with a default value of 100.

v6.0.2 – Guidewire updates the default value of column X to 200.

#### **Implementation #1 upgrade path**

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.0+++ – More custom data model changes.

v6.0.2+ – column X introduced with a default value of 200.

Result of upgrade of an entity archived at v6.0.0+: column X has value 200.

In this situation, version 6.0.1 was skipped in the upgrade path. Therefore, column X is added with the default value of 200 that it has in version 6.0.2.

#### **Implementation #2 upgrade path**

v6.0.0+ – Start with version 6.0.0 data model with custom changes.

v6.0.0++ – More custom data model changes.

v6.0.1+ – column X introduced with a default value of 100.

v6.0.2+ – column X already exists.

Result of upgrade of an entity archived at v6.0.0+: column X has value 100.

In this situation, column X is added in version 6.0.1 with the default value of 100. During the upgrade from version 6.0.1 to version 6.0.2, column X already exists. Therefore, the upgrader does not add the column. A custom trigger would be required to update the value of X from 100 to 200 during the upgrade from version 6.0.1 to 6.0.2.

## 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 PolicyCenter 8.0.3 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 PolicyCenter from sending messages until you have verified a successful database upgrade.

### To suspend message destinations

1. Start the PolicyCenter server for the pre-upgrade version.
2. Log in to PolicyCenter 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.

## Configuring the Database Upgrade

You can set parameters for the database upgrade in the PolicyCenter 8.0.3 `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 PolicyCenter to the Application Server” on page 82 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 PolicyCenter encryption is applied on one or more attributes, the PolicyCenter 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, PolicyCenter:

- 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 288.
<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 PolicyCenter 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 PolicyCenter retrieves the current state of the counters at the end of the execution of the version trigger. PolicyCenter 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 and indexes on the `ArchivePartition` column on all entities that PolicyCenter can archive. 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, PolicyCenter runs the `DeferredUpgradeTasks` work queue as soon as the upgrade completes and the server is completely started. The `DeferredUpgradeTasks` work queue creates the nonessential performance indexes and indexes on archived entities. The database user must have permission to create indexes until after the `DeferredUpgradeTasks` work queue is complete.

Deferring nonessential index creation can shorten the duration of the upgrade process. The PolicyCenter 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` work queue 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` work queue has run to completion, PolicyCenter 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.

Do not use the archiving feature until the `DeferredUpgradeTasks` work queue has completed successfully.

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

If you do not opt to defer creation of nonessential indexes, PolicyCenter 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 PolicyCenter database upgrade marks columns as unused.

To configure the PolicyCenter 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 PolicyCenter 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 PolicyCenter 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 PolicyCenter does not generate statistics. This setting enables the upgrade to update statistics on changed objects. It also configures PolicyCenter 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, PolicyCenter 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 PolicyCenter 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 44 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 `pc_someTable_EXT`, specify a `<tablestatistics>` element for that table with the `action` attribute set to `keep`:

```
<database>
  ...
  <databasestatistics>
    <tablestatistics name="pc_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. 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 tempbldb 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 315.

## 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 PolicyCenter and you can also add custom version checks.

You can configure PolicyCenter 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 PolicyCenter 8.0.3 by running the following command from the bin directory:

```
gwpc 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, PolicyCenter 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.

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

If PolicyCenter 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, PolicyCenter runs all version checks regardless of a failure in one of the checks. During a regular upgrade, PolicyCenter 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.

## Specifying ValueType on Coverage Terms

In your pre-upgrade configuration, specify a **ValueType** on all customized direct and option coverage terms in your configuration. If **ValueType** is not specified for a custom direct or option coverage term, the upgrade will fail during the configuration upgrade with the following error:

```
[java] java.lang.IllegalStateException: ValueType needs to be set on one or more cov terms
```

**ValueType** does not need to be specified for unmodified direct or option coverage terms that are provided with the default configuration.

---

**WARNING** In PolicyCenter versions prior to 8.0.1, **ValueType** is an immutable field on direct and option coverage term types. The server will not start if you change the **ValueType** after you have set it on a production mode server. You can change the **ValueType** in Studio to satisfy the requirement that all customized direct and option coverage terms have a **ValueType** set. However, you cannot start the pre-upgrade server after setting a **ValueType**. As of PolicyCenter 8.0.1, **ValueType** is no longer an immutable field on direct and option coverage term types.

---

Check all direct and option coverage terms in your custom configuration for null **ValueType** fields. For any null **ValueType** fields in your custom configuration, specify a value type.

**To specify ValueType on custom direct and option coverage terms in PolicyCenter 4.0**

1. Open Studio in your pre-upgrade configuration.
2. In the Resources pane, expand **configuration** → **Product Model** → **Policy Lines**.
3. Select a policy line.
4. Click the **Coverages** tab.
5. Click each direct and option coverage term that is customized in your configuration, nested one level under the coverage to which it applies.
6. On the Basics tab, if the **Value Type** is **<none selected>**, select a **Value Type** from the drop-down list.
7. Save your changes.
8. Repeat this process for each custom direct and option coverage term in every policy line until all direct and option coverage terms have a value type specified.

**To specify ValueType on custom direct and option coverage terms in PolicyCenter 7.0**

1. Open Studio in your pre-upgrade configuration.
2. In the Resources pane, expand **configuration** → **Product Model** → **Policy Lines**.
3. On the Basics and Coverages tab, open each customized direct and option coverage term.
4. Click each direct and option coverage term that is customized in your configuration, nested one level under the coverage to which it applies.
5. If the **Value Type** is **<none selected>**, select a **Value Type** from the drop-down list.
6. Save your changes.
7. Repeat this process for each custom direct and option coverage term in every policy line until all direct and option coverage terms have a value type specified.

## Dropping Custom Rating Worksheet Tables

The database upgrade drops tables and entities that store rating worksheets directly. As of PolicyCenter 8.0.1, rating worksheets are stored on a **WorksheetContainer** stored on a policy. The **RatingWorksheet** delegate has been removed.

If you added custom tables and entities that implement the **RatingWorksheet** delegate, you can extend the upgrade to also drop these tables and entities.

If these custom tables and entities were in production before PolicyCenter 7.0.8, first extract any worksheet data that you wish to retain.

**To drop custom rating worksheet tables during upgrade**

1. In Studio, navigate to **configuration** → **config** → **dbupgrade** and open **emerald-dbupgrade-config.properties**.
2. For each custom entity that implements the **RatingWorksheet** delegate, add the entity name and corresponding table name to **RatingWorksheet.TableAndEntity**. Use the format **tableName:entityName** and separate entries with a comma.

## 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 PolicyCenter 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 PolicyCenter 8.0.3. 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.

---

PolicyCenter 7.0 checks the length of CovTermPattern codes when the server starts. If you have a CovTermPattern code that is longer than 50 characters, the server reports a product model warning, such as:

```
Code for OptionCovTermPattern "SomeStringThatContainsAnExcessiveAmountOfCharacters" exceeds maximum length of 50
```

This issue affects PolicyCenter implementations that are integrated with ClaimCenter. ClaimCenter translates PolicyCenter CovTermPattern codes into typekeys. Typekeys must be unique and not longer than 50 characters. For PolicyCenter 7.0, Studio does not allow you to create a CovTermPattern code longer than 50 characters. Versions of PolicyCenter prior to 7.0 did not have this restriction, so you might have CovTermPattern codes longer than 50 characters in an upgraded environment.

If the PolicyCenter server reports warnings about CovTermPattern code length, and your PolicyCenter implementation is or will be integrated with ClaimCenter, contact Guidewire Support for assistance.

Guidewire requires that you use a separate mirror database for reporting. If you did not do so, then you can experience problems during a database upgrade that are severe enough to prevent the upgrade.

In particular, the Premium Accounting extension package SQL scripts create special tables in the reporting database that reporting uses for storing premium accounting accrual data. The reporting scripts use these tables, but the PolicyCenter application does not. If you created these tables in a production database, then any attempt to upgrade that production database will fail.

If you encounter this situation, move data from all pcrt\_ prefixed tables to another schema. Then, drop all pcrt\_ prefixed tables from the database that you want to upgrade before starting the server to launch the upgrade.

### 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 *PolicyCenter 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 PolicyCenter 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 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`.

## Dropping PolicyLine Columns

The database upgrade removes the following columns from pc\_policyline:

- CustomStates
- IncludedMonopolyStates
- OtherStatesOpt
- StopGapOpt

## Adding SelectedVersion Edge Foreign Key to Job

The upgrade adds a non-nullable SelectedVersion edge foreign key to Job. For jobs that have at least one non-withdrawn version, the foreign key points to the non-withdrawn version with the highest branch number. For jobs that have only withdrawn versions, the foreign key points to the withdrawn version with the highest branch number.

## Dropping SyncedToAccount Column from AccountSyncable Types

This step drops the syncedtoaccount column from tables for types that implement AccountSyncable. This includes the following entities:

- pc\_policycontactrole
- pcst\_policycontactrole
- pc\_policylocation
- pc\_policyaddress

This step also drops the syncedtoaccount column from any custom entities that you have defined to implement AccountSyncable

## Renaming UWIssueHistory Columns

The database upgrade renames the following columns on UWIssueHistory.

Old name	New name
ApprovingUser	ResponsibleUser
AutomaticApprovalCause	AutomaticOperationCause
DurationType	ApprovalDurationType

UWIssueHistory objects capture data about approval events, but they also capture data about other events. So, the columns have been renamed to not be specific to approval events.

## Merging UWIssueApproval into UWIssue

To ensure that concurrent changes to issues and approvals are detectable, PolicyCenter 4.0 merges the UWIssueApproval entity into UWIssue. UWIssueApproval is now a Gosu object with the same behavior as the previous entity.

This step populates pc\_uwissue with the contents of pc\_uwissueapproval. This step also adds the following columns to the staging table pcst\_uwissue:

- ApprovalValue
- ApprovalBlockingPoint
- ApprovalDurationType
- ApprovalExpirationDate
- ApprovingUser
- AutomaticApprovalCause
- CanEditApprovalBeforeBind
- HasApprovalOrRejection

Following the consolidation of `UWIssueApproval` entity into `UWIssue`, the database upgrade drops the `pc_uwissueapproval` and `pcst_uwissueapproval` tables.

### Setting `CanEditApprovalBeforeBind` to True for `UWIssues` in Bound `PolicyPeriods`

This step sets the `CanEditApprovalBeforeBind` flag on `UWIssue` to true for all approved issues in bound `PolicyPeriods`. The upgrade does not modify the `EditBeforeBind` flag on open `PolicyPeriods`. The upgrade logic is consistent with the logic to set this field that was introduced in PolicyCenter 4.0.2. Promoting a period sets `EditBeforeBind` to true on all approvals in the period. Editing an open job reopens any issue whose approval has `EditBeforeBind` equal to false.

This step only runs for upgrades from PolicyCenter 4.0.1 or newer, which is the version in which the `UWIssueApproval` entity was removed.

### Updating System Tools Permissions

As of PolicyCenter 4.0.2, permission to view the system tools is divided into multiple permissions, providing a greater degree of control over which roles can see which tools. PolicyCenter 4.0.2 and newer replaces the original permission, `toolsCacheInfoView`, with the following permissions:

- `toolsbatchprocessview`
- `toolsclusterview`
- `toolsinfoview`
- `toolsJMXBeansview`
- `toolslogview`
- `toolsPluginview`
- `toolsprofilerview`

This step grants the above permissions to any role that originally had the `toolsCacheInfoView` permission. After the upgrade, you can update the permissions assigned to a role if you want to more finely restrict access to the system tools.

### Converting Modifiers and Rate Factors to Delegates

In versions of PolicyCenter prior to 4.0.2, all `Modifier` entities were stored in one table with subtypes for different lines and modifiable associations. This can cause a performance issue due to the time required to maintain numerous indexes on the `pc_modifier` table.

In PolicyCenter 4.0.2, `Modifier` and `RateFactor` entities are modeled using delegates, similar to `Coverage` entities.

This step updates the data in `Modifier` and `RateFactor` tables and distributes it into appropriate tables corresponding to new entity definitions:

- The data from the `pc_modifier` table is split into separate tables corresponding to entities that implement the `Modifier` delegate using a `ModifierAdapter`. The database upgrade creates a new table for each `Modifier` subtype and moves data for the subtype from `pc_modifier` to the new table.
- The data from the `pc_ratefactor` table is split into separate tables corresponding to entities that implement the `RateFactor` delegate using a `RateFactorAdapter`.

The step handles the following configuration scenarios:

- Extension fields on `Modifier` and `RateFactor`.
- Extension fields on subtypes of `Modifier`. There are no subtypes of `RateFactor`.
- Extension subtypes of `Modifier`, including all defined fields and arrays.

The database upgrade does not handle the following scenarios:

- Subtypes of subtypes of `Modifier` or `RateFactor`.

- Child array properties defined in extensions of base `Modifier` or `RateFactor` entities. This is different from defining arrays on an extension subtype of `Modifier` or `RateFactor`. The database upgrade supports the latter situation.

If you have any of these conditions, contact Guidewire Support for assistance.

The automated configuration upgrade truncates the associated table names for both `Modifier` and `RateFactor` entities to 25 characters.

Gosu code that directly references one of the subtypes of `Modifier` or `RateFactor`, either base or an extension, does not need to be changed. The new entities have the same type names as the old subtypes.

### Setting User Role Assignment Close Date to Job Close Date

This step sets the `pc_UserRoleAssign.CloseDate` to the `pc_Job.CloseDate` for the corresponding `Job` record where `pc_UserRoleAssign.JobID` equals `pc_Job.ID`.

### Updating User Role Assignment Relationships with Policy and Job

This step updates `pc_UserRoleAssign` to have disjoint relationships with `Policy` and `Job`. The upgrade sets `pc_UserRoleAssign.PolicyID` to null for any `pc_UserRoleAssign` record with a `JobID` that is not null. A `UserRoleAssignment` record can be associated with a `Policy` or a `Job`, but not both.

### Moving Effective-Dated Registry from Branch to Container

PolicyCenter 4.0.1 included an effective-dated registry based on the effective-dated branch (for example, `PolicyPeriod`). In PolicyCenter 4.0.2, the registry is moved to the container-level (for example, `Policy`). This step moves the data from the branch registry table to the container registry table. This registry is a denormalized table added to improve performance. The database upgrade also drops the old registry table, `pc_policyperiodregistry`.

This step is only necessary for upgrades from PolicyCenter 4.0.1. There is already a step to populate the table correctly for upgrades from PolicyCenter 3.0.x to PolicyCenter 8.0.3.

### Denormalizing Contact to PolicyContactRole

This step denormalizes `Contact` to the `PolicyContactRole`.

First, the upgrade alters `pc_AccountContactRole.AccountContact` to not allow null. Any existing `AccountContactRole` records with a null `AccountContact` are corrupt data. The upgrade deletes any `AccountContactRole` records with a null `AccountContact`.

The upgrade then adds a `ContactDenorm` foreign key column to the `pc_PolicyContactRole` table and sets `ContactDenorm` to the `AccountContact.Contact` of the `PolicyContactRole.AccountContactRole`.

### Adding MostRecentTerm to PolicyTerm

PolicyCenter 4.0.3 adds a `MostRecentTerm` to `PolicyTerm`. The `MostRecentTerm` is a bit column that PolicyCenter uses to mark a `PolicyTerm` as the most recent version. This step initializes the `pc_PolicyTerm.MostRecentTerm` column.

For each term, the upgrade sets `MostRecentTerm` to 1 if:

- The first bound period has the highest term number of all bound periods in the policy.
- The period either has no `BasedOnID` or the `BasedOnID` has the highest model number of any period in its term that has some first bound period based on that period.

## Adding AccountHolderCount to Contact

The upgrade adds an `AccountHolderCount` column to `Contact` and sets `AccountHolderCount` to the number of records in the `pc_acctholderedge` where `pc_acctholderedge.ForeignEntityID` equals `pc.Contact.ID`.

## Upgrading Financials

The upgrade populates the following new financials columns in `pc_patransaction` and `pc_policyperiod`:

Column	Type
<code>pc_patransaction.tobeaccrued</code>	bit
<code>pc_policyperiod.totalcostrpt</code>	money
<code>pc_policyperiod.totalpremiumrpt</code>	money

The upgrade initially sets `pc_patransaction.tobeaccrued` to a default value of 1. The upgrade then sets `pc_patransaction.tobeaccrued` to 0 for rows in which the corresponding `pacost` has a `RateAmountType` other than `StdPremium` or `NonstdPremium`.

The upgrade then sets `pc_policyperiod.totalcostrpt` to the sum of `actualamount` values from `pc_pacost` rows where `pc_pacost.branchid` equals `pc_policyperiod.id`.

Finally, the upgrade sets `pc_policyperiod.totalpremiumrpt` to the sum of `actualamount` values from `pc_pacost` rows where `pc_pacost.branchid` equals `pc_policyperiod.id` and `pc_pacost.rateamounttype` is not `taxSurcharge`.

## Updating TableRegistryEntry and EncryptedColumnRegistryEntry to Use Lowercase

The upgrade sets all `TableName` values in the `pc_TableRegistry` and `pc_EncryptedColumnRegistry` tables to lowercase values.

## Dropping Upgrade Instrumentation Tables

The database upgrade drops the following upgrade instrumentation tables:

- `pc_purgeerror`
- `pc_purgehistory`
- `pc_purgerecord`
- `pc_upgradecuststatement`
- `pc_upgradeencryptchunk`
- `pc_upgradeencryptstep`
- `pc_upgradephase`
- `pc_upgradephasedbmsdump`
- `pc_upgradestep`
- `pc_upgradeversiontrigger`
- `pc_upgradevtstatement`

The database upgrade also drops the `pc_upgradevtdbmsdump.UpgradeVersionTriggerID` column.

## Migrating LoadErrorRow.RowCount to Larger Data Type

On SQL Server databases, the upgrade converts `LoadErrorRow.RowCount` 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 PolicyCenter for their parent entities:

- pc\_loaderrorrow\_RowNumber

### Populating Foreign Key to Policy on Job-level Activity Rows

PolicyCenter 7.0 added the ability to associate activities with policies, instead of just jobs and accounts. The **Activity** entity in PolicyCenter 7.0 and newer includes a foreign key to **Policy**. Job-level activities must have both the foreign keys to **Job** and **Policy** populated. The upgrade populates the foreign key to **Policy** for all existing rows where the foreign key to **Job** is not null. The upgrade determines the policy based on the **PolicyPeriod** with the same **JobID**.

### Populating InitialNotificationDate for Cancellations

PolicyCenter 7.0 added an **InitialNotificationDate** column to **Cancellation**, a subtype of **Job**. The **InitialNotificationDate** was introduced to:

- Identify if a cancellation job can be scheduled, or rescheduled. For example, if initial notices have been sent, then the cancellation can only be rescheduled.
- Recalculate the earliest allowed cancellation date during a reschedule.
- Determine if initial or replacement notices must be sent.

The upgrade populates the **InitialNotificationDate** column with the value of the **NotificationDate** column.

See “Rescheduling a Cancellation” on page 107 in the *New and Changed Guide*.

### Dropping Staging Tables

PolicyCenter only supports the import of zone data through staging tables. However, prior to PolicyCenter 7.0, most PolicyCenter tables were defined in the metadata as **Loadable**. Therefore, PolicyCenter would create a staging table for each table. The upgrade drops the staging tables and drops the **LoadCommandID** column from each table defined as **loadable** in prior versions, with the exception of zone data tables.

### Populating Foreign Key to Account on History Rows

PolicyCenter 7.0 added account-level history events, instead of just job history events. PolicyCenter 7.0 added a foreign key to **Account** to the **History** entity. Job-level history events are expected to have foreign keys to **Job** and **Account** populated. Consequently, the upgrade populates the foreign key to **Account** for all existing **History** rows. The upgrade determines the **AccountID** based on the **PolicyTermID** column.

See “Changes to Account History Screen and Events” on page 109 in the *New and Changed Guide*.

### Populating Advance Permissions

The upgrade populates new advance permissions, which have been split out from the more strict edit permissions. Anyone who can edit is permitted to advance. So, the upgrade grants a new advance permission for each role with the existing edit permission.

Existing permission	New permission
editaudit	advanceaudit
editcancellation	advancecancellation
editissuance	advanceissuance
editpolchange	advancepolchange
editreinstate	advanceeinstate
editrenewal	advancerenewal
editrewrite	advancerewrite
editsubmission	advancesubmission

## Populating PolicyTerm.Bound

For upgrades from versions prior to 7.0, this step populates `PolicyTerm.Bound` with `true` for promoted submission and rewrite jobs. The upgrade checks `PolicyPeriod.Status` for the `PolicyPeriod` of the `PolicyTerm`. If `PolicyPeriod.Status` equals `Bound`, then the upgrade sets the corresponding `PolicyTerm.Bound` to `true`. The `Bound` flag on the policy term signifies that the policy has been paid and is legally bound.

## Populating BusinessAutoLine.PolicyType

PolicyCenter 7.0 introduced a new column, `BusinessAutoLine.PolicyType`, a typekey to `BAPolicyType`. PolicyCenter 7.0 and newer require that `BusinessAutoLine.PolicyType` is specified, so this step populates it. The upgrade sets `BusinessAutoLine.PolicyType` to `BA`, indicating the business auto typecode for all existing rows. Business auto was the only commercial auto policy type that PolicyCenter 4.0 explicitly supported. The upgrade only populates data in rows where `PolicyType` is already null. This is to protect against the unlikely event that a custom extension used the same column name.

## Initializing PolicyPeriod.NewInvoiceStream

The upgrade initializes `PolicyPeriod.NewInvoiceStream` for quoted jobs that create a new policy term. The types of jobs that create a new policy term are:

- Renewal
- Rewrite
- RewriteNewAccount
- Submission

## Redefining Relationships Between BAJurisdiction and BAHiredAutoBasis and BANonOwnedBasis

In versions of PolicyCenter prior to 7.0, `BAJurisdiction` had foreign keys to `BAHiredAutoBasis` and `BANonOwnedBasis`. As of PolicyCenter 7.0, `BAJurisdiction` has a one-to-one relationship defined to `BAHiredAutoBasis` and `BANonOwnedBasis`. Reverse foreign keys back to `BAJurisdiction` are also defined.

The upgrade populates the foreign key to `BAJurisdiction` for every row of `BAHiredAutoBasis` and `BANonOwnedBasis`, pointing to the row of `BAJurisdiction` that points to the `BAHiredAutoBasis` or `BANonOwnedBasis`.

The upgrade then drops the `BAJurisdiction` foreign keys to `BAHiredAutoBasis` and `BANonOwnedBasis`.

## Populating CurrentNotificationsSent on Cancellations

In versions of PolicyCenter prior to 7.0, the `Cancellation.NotificationDate` was overloaded as both the notification date and an indication of whether a cancellation had been successfully scheduled or rescheduled. PolicyCenter 7.0 added a bit type column, `CurrentNotificationsSent`, to `Cancellation`. PolicyCenter 7.0 updated the cancellation process to use the new `Cancellation.CurrentNotificationsSent` column. The upgrade sets `Cancellation.CurrentNotificationsSent` to `true` for any `Cancellation` records where `Cancellation.NotificationDate` is not null.

See “Rescheduling a Cancellation” on page 107 in the *New and Changed Guide*.

## Adding Jurisdiction Typelist

PolicyCenter 7.0 introduced a new `Jurisdiction` typelist, as an addition, but not a replacement, of the `State` typelist.

The upgrade creates the `Jurisdiction` typelist table if it does not already exist. The upgrade then copies values from the `State` typelist to the `Jurisdiction` typelist, preserving IDs for the same typecodes.

See “Modifications to Typelists” on page 120 in the *New and Changed Guide*.

## Moving SecondaryNamedInsured from PersonalAutoLine up to EffectiveDatedFields

In PolicyCenter 7.0, the SecondaryNamedInsured edge foreign key on the PersonalAutoLine was replaced with a foreign key on the EffectiveDatedFields. The foreign key is on the EffectiveDatedFields because that is where revisioned policy period-level data is stored. It is a foreign key because there are no relationship graph cycles and therefore no need for an edge foreign key. As a result the implicit edge table no longer exists.

Domain methods accessing and operating on the secondary named insured have been moved from the PersonalAutoLine onto the PolicyPeriod. These domain methods were mostly defined in enhancements. As of PolicyCenter 7.0, these domain methods are on the PolicyPeriod, and they are split between enhancements and Java code, necessary in Java for some builders.

## Populating Account.LinkContacts

The upgrade sets the LinkContacts column on Accounts to true when there is at least one bound Policy on the Account. In PolicyCenter, Account.LinkContacts indicates whether an Account synchronizes Contacts with an external contact management system.

## Updating Account Synchronizable Columns

The upgrade updates account synchronizable columns with synchronized values. Storage of synchronized values changed from PolicyCenter 4.0 to PolicyCenter 7.0. In PolicyCenter 4.0, account synchronizable denormalized fields were typically populated after a policy was quoted, to lock down the value. At other times, the fields were allowed to be null. PolicyCenter 7.0 and newer support past-dated and future-dated contact changes that require population of the denormalized fields much earlier during the lifecycle of a policy job.

## Populating AccountContact.LastUpdateTime

PolicyCenter 7.0 and newer track the date and time of the most recent update to contact information. This information is captured in a LastUpdateTime column added to AccountContact, Address, and Contact entities.

The upgrade copies the value of AccountContact.UpdateTime to AccountContact.LastUpdateTime.

## Adding DateOfBirthInternal and MaritalStatusInternal to PolicyContactRole

The upgrade adds and populates the DateOfBirthInternal and MaritalStatusInternal columns to the PolicyContactRole. These columns are set to the value of the DateOfBirth and MaritalStatus columns on the corresponding Contact entity.

## Populating ParticipatingPlan.WorkersCompLine

The upgrade adds and populates ParticipatingPlan.WorkersCompLine with a foreign key to PolicyLine. The upgrade then drops PolicyLine.ParticipatingPlanID.

## Renaming UWIssue and UWIssueHistory Column ApprovalExpirationDate to ApprovalInvalidFrom

The upgrade changes ApprovalExpirationDate to ApprovalInvalidFrom on the UWIssue and UWIssueHistory tables. The upgrade renames the columns and increments each column by one day.

See “Change to End Date Field in Underwriting Entities” on page 113 in the *New and Changed Guide*.

## Moving MotorVehicleRecord from Base Metadata to Extensions

In PolicyCenter 7.0, the MotorVehicleRecord entity was deprecated. The upgrade renames pc\_MotorVehicleRecord to pcx\_MotorVehicleRecord to be consistent with the suggested naming of extensions.

See “Deprecation of MotorVehicleRecord” on page 117 in the *New and Changed Guide* and “Motor Vehicle Records in Personal Auto” on page 106 in the *New and Changed Guide*.

## Dropping Entities for Business Auto Line of Business

The upgrade drops the `BADriveOtherCarCov` and `BAOtherDriver` entities and the `BAStateCov.OtherDrivers` column.

## Erasing Database-based Archiving

The upgrade removes tables and columns used for archiving from the database. The upgrade drops the following tables and columns:

- `pc_ArchiveAdminKey`
- `pc_ArchiveGraphRecord`
- `pc_ArchiveTransitionRec`
- `pc_ArchiveTypeKey`

The database archiving feature was not available for PolicyCenter.

## Removing InetSoft Reporting Support

The upgrade removes the following database elements that were involved in supporting InetSoft reporting:

- `pc_reportgroup`
- `pc_rolerptprivilege`
- `pc_rptgroup rpt`
- `pc_sreereport`

The upgrade also drops the `pc_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 PolicyCenter does not resend notifications.

## Setting Parameter for Data Files Imported

The upgrade sets the `data_files_imported` parameter to `finished` in the `pc_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 PolicyCenter.

## Setting IndividualStacks column on WorkQueueProfilerConfig to Non-nullable

The upgrade sets the `IndividualStacks` column on `pc_WorkQueueProfilerConfig` to non-nullable.

## Changing Contact Foreign Keys on ContactAutoSyncWorkItem

This is a Guidewire platform-level upgrade step. Because PolicyCenter does not use the Contact Auto Sync work queue, this step does not affect PolicyCenter.

The upgrade first checks that the `pc_ContactAutoSyncWorkItem` table is empty. Then the upgrade drops the `minContactID` and `maxContactID` foreign keys to `pc_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 `pc_Activity` to `PK_Activity`. On SQL Server, the upgrade renames the primary key index on `pc_Activity` to `pc_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:

- `pc_ArrayDataDist`
- `pc_ArraySizeCntDD`
- `pc_AssignableForKeyDataDist`
- `pc_AssignableForKeySizeCntDD`
- `pc_BeanVersionDataDist`
- `pc_BlobColDataDist`
- `pc_BooleanColDataDist`
- `pc_ClobColDataDist`
- `pc_DateAnalysisDataDist`
- `pc_DateBinnedDDDateBin`
- `pc_DateBinnedDDValue`
- `pc_ForKeyDataDist`
- `pc_GenericGroupCountDataDist`
- `pc_HourAnalysisDataDist`
- `pc_LoadInsertSelect`
- `pc_LoadOperation`
- `pc_LoadRowCount`
- `pc_NullableColumnDataDist`

- pc\_TableDataDist
- pc\_TableUpdateStats
- pc\_TypecodeCountDataDist
- pc\_TypekeyDataDist

### [Adding CPBlanketAutoNumber to Commercial Property Policy Lines](#)

For all Commercial Property policy lines, the upgrade creates a new `AutoNumberSequence` instance per policy term and updates `CPBlanketAutoNumberSeq` to point to the `AutoNumberSequence`. `PolicyLine` rows that correspond to the same policy term point to the same sequence.

This step applies to upgrades from versions prior to 7.0.1.

### [Dropping AccountAccount Staging Table](#)

The upgrade drops the `pcst_AccountAccount` staging table and removes `pc_AccountAccount.LoadCommandID`. The `AccountAccount` entity was introduced in PolicyCenter 7.0.0.

This step applies to upgrades from versions prior to 7.0.1.

### [Dropping FormPatternCovTermValue Name and Description Columns](#)

The upgrade drops the `Name` and `Description` columns from `FormPatternCovTermValue`. PolicyCenter derives coverage term value names and descriptions directly from the product model, so names and descriptions do not need to be stored with the form patterns.

This step applies to upgrades from versions prior to 7.0.1.

### [Dropping RIRiskCombination](#)

The upgrade drops the `pc_RIRiskCombination` table. This table only existed in PolicyCenter 7.0.0 and is not being used in PolicyCenter 7.0.1.

This step applies to upgrades from versions prior to 7.0.1.

### [Moving PolicyAddress from PolicyPeriod to EffectiveDatedFields](#)

The upgrade moves the `PolicyAddress` value for each `PolicyPeriod` record to the corresponding `EffectiveDatedFields` entity.

This step applies to upgrades from versions prior to 7.0.1.

### [Resynchronizing Product Model Lookups](#)

The upgrade deletes checksums for product model lookups from the database, causing PolicyCenter to resynchronize product model files when the server starts. This is to resolve an issue with determining the appropriate availability lookup row in certain situations.

This step applies to upgrades from versions prior to 7.0.1.

### [Dropping ArchiveID from PolicyPeriod](#)

The upgrade drops the `pc_PolicyPeriod.ArchiveID` column. This column was not used by PolicyCenter.

This step applies to upgrades from versions prior to 7.0.1.

## Dropping FormPatternCovTerm Name and Description Columns

The upgrade drops the `Name` and `Description` columns from `FormPatternCovTerm`. PolicyCenter derives coverage term names and descriptions directly from the product model, so names and descriptions do not need to be stored with the form patterns.

This step applies to upgrades from versions prior to 7.0.1.

## Moving RIRiskNumber and Dropping RIRiskVLContainer

The upgrade copies the risk number from the `RIRiskVLContainer` to each associated `RIRiskVersionList` and deletes the `pc_RIRiskVLContainer` table.

This step applies to upgrades from versions prior to 7.0.1.

## Adding Denormalized Primary Address Columns to Contact

The upgrade adds denormalized primary address columns on `pc_Contact` and populates these columns with values from `pc_Address`. This is done to improve performance of account searches when minimal name and any of the minimal primary address fields are specified. This step adds and populates the following columns on `pc_Contact`:

- `City`
- `State`
- `PostalCode`
- `Country`
- `CityDenorm`
- `PostalCodeDenorm`

PolicyCenter keeps these columns synchronized with their corresponding values on `pc_Address`. PolicyCenter also includes a database consistency check that checks that the `pc_Contact` and `pc_Address` values are synchronized.

This step applies to upgrades from versions prior to 7.0.2.

## Dropping Extractable and OverlapTable from AutoNumberSequence

The upgrade drops `Extractable` and `OverlapTable` delegate columns from `AutoNumberSequence`. The upgrade drops the `admin` and `archivepartition` columns from `pc_AutoNumberSequence`.

This step applies to upgrades from versions prior to 7.0.3.

## Setting Referenced Column on AccountContactRole

The upgrade sets the `Referenced` Boolean column on `pc_AccountContactRole` to `true` if there is any locked policy period with a `PolicyContactRole` referring to the `AccountContactRole`.

This step applies to upgrades from versions prior to 7.0.3.

## Setting Referenced Column on AccountLocation

The upgrade sets the `Referenced` Boolean column on `pc_AccountLocation` to `true` if there is any locked policy period with a `PolicyLocation` referring to the `AccountLocation`.

This step applies to upgrades from versions prior to 7.0.3.

## Setting Referenced Column on Address

The upgrade sets the `Referenced` Boolean column on `pc_Address` to `true` if there is any locked policy period with a `PolicyAddress` referring to the `Address`.

This step applies to upgrades from versions prior to 7.0.3.

### Dropping CoverageSymbolGroup from Coverage

The upgrade drops the `CoverageSymbolGroup` column from `pc_Coverage`. The upgrade first checks that entities that implement the `Coverage` delegate have no data in their `CoverageSymbolGroup` foreign keys.

The `CoverageSymbolGroup` foreign key has been removed from the `Coverage` entity and replaced with an enhancement property. Before upgrading, `CoverageSymbolGroup` must be nulled out. This only affects Business Auto coverages.

Additionally, it is possible, though very unlikely, to have configured entity extensions to have a similar issue to the one described above with `Coverage`. If that has occurred, an error will be reported during the upgrade process and the data will need to be fixed before continuing.

This step applies to upgrades from versions prior to 7.0.3.

### Dropping City and PostalCode from Contact

The upgrade drops unnecessary `PrimaryAddress` denormalized columns `City` and `PostalCode` from `pc_Contact`.

This step applies to upgrades from versions prior to 7.0.3.

### Populating PNIContactDenorm on PolicyPeriod

The upgrade adds a `PNIContactDenorm` column to `pc_PolicyPeriod`. The upgrade then populates `pc_PolicyPeriod.PNIContactDenorm` with the primary named insured's contact on the policy.

This step applies to upgrades from versions prior to 7.0.3.

### Replacing Index on CalcRoutineDefinition

The upgrade drops the index on `Code`, `Version`, and `Retired` on `pc_CalcRoutineDefinition`. A new index is defined in `CalcRoutineDefinition.etcx`. The new index adds the `Jurisdiction` column to `Code`, `Version`, and `Retired`. By adding the `Jurisdiction` column, multiple `CalcRoutineDefinitions` with the same `Code` and `Version` can exist.

This step applies to upgrades from versions prior to 7.0.4.

### Replacing RateTableDataType Double Typecode with Decimal

The upgrade replaces the `Double` typecode on `pct1_RateTableDataType` with a `Decimal` typecode.

This step applies to upgrades from versions prior to 7.0.4.

### Rename RateBook.BookVersion to RateBook.BookEdition

The upgrade renames `pc_RateBook.BookVersion` to `pc_RateBook.BookEdition`.

This step applies to upgrades from versions prior to 7.0.4.

### Dropping Form-related PMLockedEntity and PMLockedFields Entries

The upgrade drops form-related `PMLockedEntity` and `PMLockedField` entries, including:

- `FormPattern`
- `ProductForm`
- `ProductPattern`

- PolicyLineForms
- ProductFormPattern
- ProductForms

This step applies to upgrades from versions prior to 7.0.5.

### [Populating Default Code for All Rate Table Operands](#)

The upgrade adds an ArgumentSourceSetCode column to pc\_CalcStepDefOperand and sets the value of ArgumentSourceSetCode to DEFAULT for records where TableCode is not null.

This step applies to upgrades from versions prior to 7.0.5.

### [Deleting Impact Analysis Test Prep Batch Process](#)

The upgrade deletes the Impact Analysis Test Prep batch process and the related tables pc\_ImpactAnalysisPeriod and pc\_ImpactAnalysisCase.

This step applies to upgrades from versions prior to 7.0.5.

### [Dropping Description from CalcStepDefinition](#)

The upgrade drops the pc\_CalcStepDef.Description column.

This step applies to upgrades from versions prior to 7.0.6.

### [Converting CalcStepDefinition Columns from Text to MediumText](#)

The upgrade first checks that the Notes and SectionComment columns on pc\_CalcStepDef can be truncated to fit in a mediumtext type column. If any rows have columns that are too long for mediumtext, the upgrade reports an error and provides an SQL query to find the rows. If all rows can be truncated, the upgrade converts Notes and SectionComment to mediumtext.

This step applies to upgrades from versions prior to 7.0.6.

### [Populating the Business Vehicle Foreign Key on BACost](#)

For BusinessVehicleCovCost subtypes of BACost, the upgrade populates the BusinessVehicle foreign key with the corresponding Vehicle from BusinessVehicleCov.

This step applies to upgrades from versions prior to 7.0.6.

### [Setting RateTableDefinition EntityName Values where Null](#)

The upgrade sets each null EntityName on RateTableDefinition to the value RateFactorRow.

This step applies to upgrades from versions prior to 7.0.6.

### [Populating PolicyTermArchiveState on PolicyTerm](#)

This upgrade step converts PolicyTerm.Archived to PolicyTerm.PolicyTermArchiveState. The upgrade queries the database for all PolicyPeriod records that have foreign keys to a PolicyTerm.

If all PolicyPeriod records for a PolicyTerm have a non-null ArchiveState, then the upgrade sets PolicyTermArchiveState to TC\_FULLYARCHIVED.

If all PolicyPeriod records for a PolicyTerm have a null ArchiveState, then the upgrade leaves PolicyTermArchiveState set to TC\_NOTARCHIVED. This is the default value of PolicyTermArchiveState.

If the `PolicyPeriod` records for a `PolicyTerm` have a mix of null and not null `ArchiveState` values, then the upgrade sets `PolicyTermArchiveState` to `TC_PARTIALLYARCHIVED`.

This step applies to upgrades from versions prior to 7.0.7.

### Populating Attachment Inclusion

The upgrade creates a `pc_RIAttachmentInclusion` table if the table does not yet exist. If the table exists, but does not include a `PolicyTerm` column, the upgrade creates the `PolicyTerm` column. The upgrade then segments inclusions by policy term and makes inclusions retireable.

### Regenerating InstrumentedWorker and Dropping InstrumentedWorkerTask

The database upgrade drops the `pc_InstrumentedWorker` and `pc_InstrumentedWorkerTask` tables. The `pc_InstrumentedWorker` table is regenerated when PolicyCenter starts and includes data model changes that Guidewire made to the table between versions. The `pc_InstrumentedWorkerTask` table replaces the `pc_InstrumentedWorkerTask` table used in versions prior to PolicyCenter 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:

- `BusinessWeek`
- `Region`
- `Role`
- `UWAuthorityProfile`

### Dropping ClusterInfo

The upgrade drops the `pc_ClusterInfo` table. This table is renamed `pc_BatchServer`. The new table is created following the upgrade when PolicyCenter starts. The `pc_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.

PolicyCenter 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 `pc_ClusterMemberData` table.

### Updating SpatialPoint on Address

The upgrade updates the `SpatialPoint` column on `pc_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 `pc_ProcessHistory`.

### Dropping WorkQueueName Column from WorkQueueWorkerControl

The upgrade drops the `pc_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 `pc_Activity`, `pc_Address`, and `pc_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 `pc_dbConsistCheckRun`.
- drops the `Subtype` column from `pc_dbConsistCheckQueryExec`.
- updates null values of the `TableName` and `ThreadName` columns of `pc_dbConsistCheckQueryExec` to the value `UNKNOWN`.

## Upgrading Database Statistics Tables

The upgrade makes the following changes to tables involved in gathering database statistics for PolicyCenter:

- deletes all `pc_ProcessHistory` records for the Incremental Database Statistics process.
- drops the `Subtype` column from `pc_DatabaseUpdateStats`, `pc_TableUpdateStats`, and `pc_TableUpdateStatsStatement`.
- sets null values for `pc_TableUpdateStatsStatement.ObjectName` and `pc_TableUpdateStatsStatement.UpdateStatsStatement` to `UNKNOWN`.
- drops `pc_DatabaseUpdateStats.NumThreads`.
- drops `pc_TableUpdateStats.Deletes`, `pc_TableUpdateStats.Inserts` and `pc_TableUpdateStats.RowCount`.

## Dropping Upgrade-related Tables

The upgrade drops the following tables:

- `pc_UpgradeDBParameterPair`
- `pc_UpgradeDBParameterRow`
- `pc_UpgradeDBParameterSet`

## Dropping AddressBookFingerprint from Contact and ContactCategoryScore

The upgrade drops the `AddressBookFingerprint` column from `pc_Contact` and `pc_ContactCategoryScore`. The upgrade also drops the `AddressBookFingerprint` property from the `Contact` and `ContactCategoryScore` entities.

## Populating Original Effective Date of Policy

The upgrade creates an `OriginalEffectiveDate` column on `pc_Policy` and populates the column with the start date of the policy. The upgrade determines the start date of the policy by finding the earliest `PeriodStart` value in `pc_PolicyPeriod` for the matching `policyID` with a `PolicyPeriodStatus` of `Bound`.

## Upgrading Currency

The database upgrade runs several steps that upgrade currency values.

### Upgrading Contact Table

This step upgrades the pc\_Contact table by correctly repopulating the AccountHolderCount field and setting a default currency type for PreferredSettlementCurrency.

### Populating Preferred Coverage and Settlement Currency on PolicyPeriod

The upgrade populates PreferredCoverageCurrency and PreferredSettlementCurrency on pc\_PolicyPeriod with the default Currency.

### Adding Monetary Amount Currency Fields

The upgrade creates a currency field for all MonetaryAmounts and populates this field with the default currency.

### Populating Default Currencies on Account

The upgrade populates the PreferredCoverageCurrency and PreferredSettlementCurrency columns on Account to the default currency.

### Populating Default Currency Fields on RIAGreement and RIProgram

The upgrade populates the Currency fields of RIAGreement and RIProgram with the default currency. The upgrade also populates the ReinsuranceCurrency column on Reinsurable with the default currency.

### Populating ProducerCodeCurrency with Default Currency for each ProducerCode

The upgrade creates a ProducerCodeCurrency table with ProducerCodeId and Currency columns. The upgrade then populates ProducerCodeId with the corresponding ID from the ProducerCode table and populates Currency with the default currency.

### Populating Billing Amounts for Costs and Transactions

For Cost and delegates of Cost, the upgrade adds and populates the following columns:

- ActualAmountBilling
- ActualTermAmountBilling
- OverrideAmountBilling
- OverrideTermAmountBilling
- StandardAmountBilling
- StandardTermAmountBilling
- ActualAmountBillingCurrency
- ActualTermAmountBillingCurrency
- OverrideAmountBillingCurrency
- OverrideTermAmountBillingCurrency
- StandardAmountBillingCurrency
- StandardTermAmountBillingCurrency
- ActualAmountCurrency
- ActualTermAmountCurrency
- OverrideAmountCurrency
- OverrideTermAmountCurrency
- StandardAmountCurrency
- StandardTermAmountCurrency

The upgrade populates the currency columns with the default currency in rows where the corresponding amount column is not null.

The upgrade populates the settlement billing columns with the value of the base column. For example, `ActualAmountBilling` is set to the value of `ActualAmount`.

For `Transaction` and delegates of `Transaction`, the upgrade adds and populates the following columns:

- `AmountBilling`
- `AmountBillingCurrency`
- `AmountCurrency`

The upgrade populates these columns in the same manner as the `Cost` columns.

#### **Populating Default Currency on Delegates**

The upgrade adds a `Currency` column and populates it with the default currency for entities that implement the following delegates:

- `Coverable`
- `Coverage`
- `Exclusion`
- `PolicyCondition`

#### **Adding and Populating LastNotifiedCancellationDate on Cancellation**

The upgrade sets the `LastNotifiedCancellationDate` on `Cancellation` to the `CancellationDate` of the `PolicyPeriod` referenced by `SelectedVersion` if the `InitialNotificationDate` of the `Cancellation` is not null.

#### **Splitting UserRoleAssignment Table**

The upgrade splits `pc_UserRoleAssign` into `AccountUserRoleAssign`, `JobUserRoleAssign`, and `PolicyUserRoleAssign` tables. The upgrade moves records from `pc_UserRoleAssign` to the corresponding new table depending on whether `AccountID`, `JobID`, or `PolicyID` is not null. For example if a record has a non-null `AccountID`, the record is moved to the `AccountUserRoleAssign` table.

#### **Dropping UserGroupStats**

The upgrade drops the `UserGroupStats` table.

#### **Renaming UserStats Batch Process to TeamScreens**

The upgrade changes the `TypeCode` of the `UserStats` record in the `pct1_BatchProcessType` table to `TeamScreens`.

#### **Populating DisplayText on RateTableMatchOp**

The upgrade populates the `DisplayText` column on `RateTableMatchOp` with the value of the `Name` column.

#### **Changing PolicyDriver.LicenseState and OfficialID.State to Jurisdiction IDs**

The upgrade modifies `pc_PolicyDriver.LicenseState` and `pc_OfficialID.State` values from type `State` to type `Jurisdiction`.

#### **Checking for Successful Loading of emerald-dbupgrade-config.properties**

The upgrade attempts to load properties from `emerald-dbupgrade-config.properties`. If the upgrade fails to read the file or load properties, it reports an error and stops the upgrade.

## Changing PriorPolicy to Extendable

PriorPolicy is an extendable class in PolicyCenter 8.0. The upgrade adds a Subtype column to pc\_PriorPolicy and sets Subtype for all records to PriorPolicy.

## Renaming Deferred Upgrade Batch Process

The upgrade renames the DeferredUpgrade batch process type to DeferredUpgradeTasks.

## Truncating pc\_Dynamic\_Assign

The upgrade truncates the pc\_Dynamic\_Assign table.

## Dropping pc\_t1\_Template

The upgrade drops the pc\_t1\_Template table.

## Dropping Columns from WorkItem Tables

The upgrade drops the AvailableSince and LastUpdateTime columns from all pc\_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.

## Dropping Contact.CityKanjiDenorm

The upgrade drops the pc\_Contact.CityKanjiDenorm column if it exists.

## Creating AgencyBillPlan Records

For each record in pc\_Organization with a non-null AgencyBillPlanID, the upgrade creates a record in pc\_AgencyBillPlan with the organization ID and AgencyBillPlanID.

## Dropping Rating Worksheet Tables

The upgrade drops tables and entities that store RatingWorksheets directly. As of PolicyCenter 8.0.1, rating worksheets are stored on a WorksheetContainer stored on a policy. The RatingWorksheet delegate has been removed.

## Moving CommissionPlanID from ProducerCode to ProducerCodeCurrency

The upgrade copies the pc\_ProducerCode.CommissionPlanID to pc\_ProducerCodeCurrency.CommissionPlanID and then deletes pc\_ProducerCode.CommissionPlanID.

## Deleting Checksums for Product Model Lookups

The upgrade deletes checksums for product model lookups from the database. This forces PolicyCenter to resynchronize the database with product model files.

## Moving PolicyPeriod.NewInvoiceStream.Selected to PolicyPeriod.CustomBilling

The upgrade moves and renames the `Selected` column from `PolicyPeriod.NewInvoiceStream.Selected` to `PolicyPeriod.CustomBilling`. The `Selected` column did not indicate that the user selected to send a new invoice stream to BillingCenter. Instead, the user indicated custom billing and PolicyCenter either sends an invoice stream or modifies an existing one. PolicyCenter did not send any invoice stream information otherwise. Therefore the column has been renamed to `PolicyPeriod.CustomBilling` to better reflect its purpose.

## Creating the SelectedPaymentPlan PaymentPlanSummary

In PolicyCenter 8.0.2, the relationship between `PolicyPeriod` and `PaymentPlanSummary` was changed to a one-to-one relationship. The upgrade first checks that each non-retired `PaymentPlanSummary` record has a unique combination of `PolicyPeriod` and `BillingId`. The upgrade reports an error if it finds `PaymentPlanSummary` records with matching `PolicyPeriod` and `BillingId`. If the upgrade reports this error, either retire or remove the duplicate rows and restart the upgrade. When a record is restored from the archive it is upgraded to the current version. If a duplicate `PaymentPlanSummary` is detected during restoration, PolicyCenter marks the duplicate `PaymentPlanSummary` as retired.

If the upgrade finds no errors, it creates a new `SelectedPaymentPlan PaymentPlanSummary` for all `PolicyPeriods` and removes obsolete `PaymentPlanSummary` types.

## Deleting Checksums for Product Model Lookups and Lookup Rows

PolicyCenter 8.0.3 has a restructured product model. The upgrade deletes checksums for product model lookups and lookup rows from the database to allow resynchronization of product model files when the server starts.

The upgrade deletes all rows from the following tables:

- `pc_CondLookup`
- `pc_CovLookup`
- `pc_CovTermLookup`
- `pc_CovTermOptLookup`
- `pc_CovTermPackLookup`
- `pc_Exc1Lookup`
- `pc_ModifierLookup`
- `pc_OfferingLookup`
- `pc_ProductLookup`
- `pc_ProductModifierLookup`
- `pc_ProdRateFactorLookup`
- `pc_RatingFactorLookup`
- `pc_QuestionLookup`
- `pc_QuestionSetLookup`

## Viewing Detailed Database Upgrade Information

PolicyCenter 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 PolicyCenter server if it is not already running.
2. Log in to PolicyCenter 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 PolicyCenter 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. PolicyCenter 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
```

## Reloading Rating Sample Data

If you are using any rating data provided by Guidewire, such as calcRoutines, rateBooks, rateTableDefinition, parameter sets, and so forth, remove all existing rating data, and reload new rating data.

**To reload rating sample data**

1. Start the PolicyCenter server.
2. Remove the old rating sample data by running the following Gosu script against the database:

```
uses gw.transaction.Transaction
uses gw.api.database.Query

function findEntity<T extends KeyableBean>() : List<T>{
    var q = Query.make(T)
    q.startsWith("PublicID", "pc:", false /*ignoreCase*/)
    return q.select().toList()
}

Transaction.runWithNewBundle(\ bundle -> {
    findEntity<RateBook>().each(\ rb -> bundle.add(rb).remove())
    findEntity<RateTableDefinition>().each(\ rt -> bundle.add(rt).remove())
    findEntity<RateTableMatchOpDefinition>().each(\ rb -> bundle.add(rb).remove())
    findEntity<RateFactorRow>().each(\ rb -> bundle.add(rb).remove())
    findEntity<CoverageRateFactor>().each(\ rb -> bundle.add(rb).remove())
    findEntity<CalcRoutineDefinition>().each(\ rb -> bundle.add(rb).remove())
    findEntity<CalcRoutineParameterSet>().each(\ rb -> bundle.add(rb).remove())
}, "su")
```

3. Run the following Gosu script to reload the PolicyCenter 8.0.3 rating sample data:

```
uses gw.transaction.Transaction
uses gw.sampledata.small.SmallSampleRatingData
uses gw.sampledata.tiny.TinySampleRatingData

Transaction.runWithNewBundle(\ bundle -> {
    var tinyData = new TinySampleRatingData()
    tinyData.load()
    var sampleData = new SmallSampleRatingData()
    sampleData.load()
}, "su")
```

## 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 PolicyCenter 8.0.3. 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 262. 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 PolicyCenter 8.0.3 server by navigating to `PolicyCenter/bin` and running the following command:  
`gwpc dev-start`
  - b. Open a browser to PolicyCenter 8.0.3.
  - 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 PolicyCenter 8.0.3. 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 27 in the *Installation Guide* for instructions to configure the database account.
3. Install a new PolicyCenter 8.0.3 development environment. Connect this development environment to the new database account that you created in step 2. See the *PolicyCenter 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 PolicyCenter 8.0.3 development server by navigating to `PolicyCenter/bin` and running the following command:  
`gwpc dev-start`
  - b. Open a browser to PolicyCenter 8.0.3.
  - 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

PolicyCenter 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.

PolicyCenter 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 PolicyCenter or restored from the archive.

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 PolicyCenter 7.0 and PolicyCenter 8.0 phone numbers. The `parsePhoneNumber` method parses a string into a `GWPhoneNumber` object if possible. `GWPhoneNumber` is an interface that defines a standard PolicyCenter 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 `false`, 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 PolicyCenter 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 PolicyCenter. Phone numbers in PolicyCenter may become out of sync with ContactManager while the ContactManager Phone Number Normalizer work queue is running. It is safe to sync contacts in PolicyCenter that become out of sync with ContactManager. When you run the Phone Number Normalizer work queue for PolicyCenter, 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 database phone search and 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.

Wait for the Phone Number Normalizer work queue to complete before refreshing the Solr index.

## 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 265
- “Checking Database Consistency” on page 266 including “Checking that Contacts Have Unique Addresses” on page 320
- “Creating a Data Distribution Report” on page 266
- “Generating Database Statistics” on page 267. You can defer generating database statistics until your next scheduled maintenance window. You do not need to generate database statistics before using the upgraded PolicyCenter in a production environment.
- “Backing up the Database After Upgrade” on page 320

### Checking that Contacts Have Unique Addresses

An Address cannot be shared by more than one Contact. PolicyCenter 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 PolicyCenter 8.0.3 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 Integrations and Gosu from 4.0.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 323
- “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 the 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 43 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"><li>• <code>getDocumentContentsInfoForExternalUse()</code></li><li>• <code>isInboundAvailable()</code></li><li>• <code>isOutboundAvailable()</code></li></ul>	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)		<p>Understand the changes and what the code is trying to do and modify your code accordingly.</p> <p>Review database upgrade triggers to understand data model changes.</p> <p>Review the <i>New and Changed Guide</i></p>

## 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 Integrations and Gosu from 4.0.x" on page 321
<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 154 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 131 in the <i>New and Changed Guide</i>
<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 131 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 131 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 132 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 132 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 133 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 133 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 134 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 135 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 136 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 136 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code for changes to array casts.	"Array Casts" on page 137 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 137 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 138 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 138 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 139 in the <i>New and Changed Guide</i>

<input checked="" type="checkbox"/> Tasks	For more information...
<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 140 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review code to remove any Gosu interceptors.	"Interceptors Removed" on page 141 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.

<input checked="" type="checkbox"/> Tasks	For more information...
<input type="checkbox"/> Review changes to the Submission API web service and change your implementations accordingly	"SubmissionAPI Changes" on page 151 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Review changes to the Policy Renewal API web service and change your implementations accordingly	"PolicyRenewalAPI Changes" on page 152 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.

<input checked="" type="checkbox"/> 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 153 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 134 in the <i>New and Changed Guide</i>
<input type="checkbox"/> Replace deprecated contact system plugin with the latest version with a different name.	"Changes to PolicyCenter Contact-related Plugins" on page 155 in the <i>New and Changed Guide</i>